sinclair

QL

QL Archive

by Dick de Grandis-Harrison *(Psion Limited)*

CHAPTER 1
ABOUT QL ARCHIVE

QL Archive is a database program which enables you to create filing systems for any type of information you choose. You are free to decide how this information will be stored and retrieved.

You will quickly discover how Archive can be used for creating simple card index systems such as address lists or customer records. Once you have mastered the creation of straightforward systems such as these, you may wish to develop more complex multi-file relational systems where information is shared between, for example, purchase and stock control records.

Information may be presented using the screen layout that Archive provides, or you may design your own. Printed forms and reports can be produced from the information in the file in any format you choose.

One of the most powerful features of Archive is its command structure. Once you have created a file and stored some records in it, these commands can be used to find particular records, make searches and selections or display the information in the file in a particular order.

The commands combine to form a powerful programming language, similar to SuperBASIC, which can be used to construct a multitude of specialist applications.

At all times you will be guided by an informative set of prompt messages which never leave you in doubt about what your options are or what you are expected to do. If you require further information you can use the Help files. You may ask for Help at any stage, no matter what you are doing, and will automatically be given the information that is most relevant to your current needs.

The real power of Archive becomes apparent when you write your own procedures in the command language. You can create a named procedure to do exactly what you want and then use it as an additional command, in the same way as you use the commands provided with Archive.

The mechanics of writing and modifying a program are aided by a full *procedure editor* which, together with the *input line editor* (which is available at all times), make editing a simple and painless task.

The data files themselves use variable length fields and records. Not only does this lead to the most efficient use of available memory and cartridge space, but also to simplified file creation. You never need to decide in advance how large a record needs to be.

This manual contains a number of working examples. Try these out to see some of the range of things that can be done. They contain many general purpose procedures which you might include in your own programs.

If, at any time, you are not sure what to do, remember that you can ask for Help by pressing **F1**. Also remember that you can cancel any partially completed operation (e.g. typing in a number, or using a command) by pressing **ESC**.

Archive has been designed to give you the greatest possible flexibility. As a consequence it cannot give as much assistance with the selection of options as the other QL programs. If you are not familiar with computers and computer programming you may find it helpful to read the Beginner's Guide to SuperBASIC before attempting to write Archive programs.

CHAPTER 2
GETTING STARTED

LOADING QL ARCHIVE

Load QL Archive as described in the Introduction to the QL Programs. When loaded Archive will display the following message:

```
LOADING QL ARCHIVE
version x.xx
Copyright © 1984 PSION SYSTEMS
database
```

where x.xx represents the version number, e.g. 2.00.

The program will then wait for a few seconds before starting.

The Help information is not loaded into the computer's memory together with the program. It is only read from the Archive cartridge when it is needed. **You should therefore not remove the Archive cartridge from Microdrive 1 if you intend to use the Help facility.**

GENERAL APPEARANCE

When you have loaded Archive the screen should look like Figure 2.1. This is the *main display.*

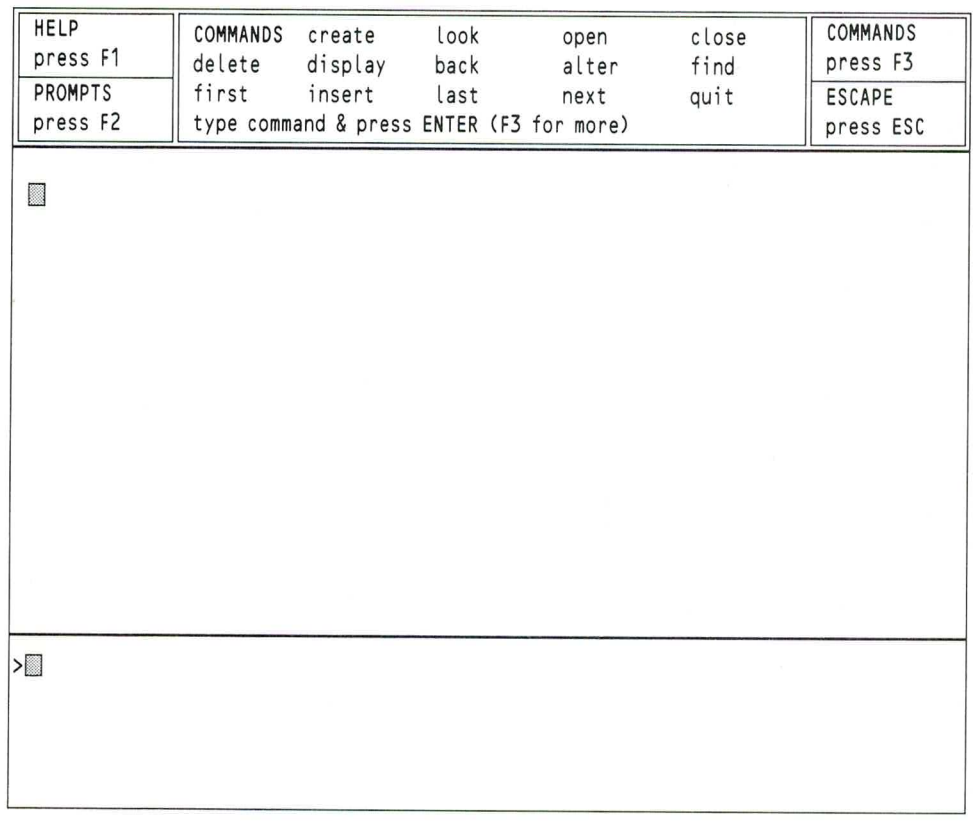

Figure 2.1 The main display with a monitor. (80 characters)

If you are using a domestic television, the screen is arranged slightly differently. This is because a television is not normally able to show clearly 80 characters per line. Archive therefore only shows 64 characters.

The screen is divided into three sections: the display area, the work area and the control area.

The Display and Work Areas

As its name suggests, this is where all information produced by Archive is shown.

The work area uses the bottom four lines of the screen. All commands that you type in, together with any error messages, are shown here.

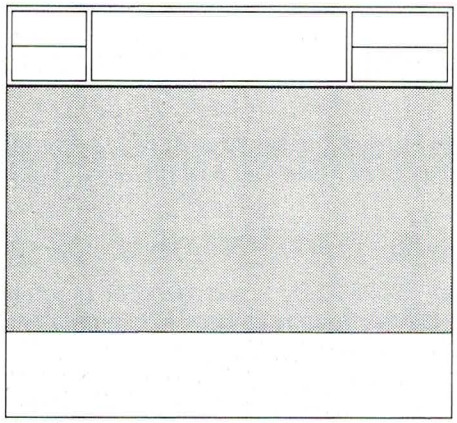

Figure 2.2 The display area

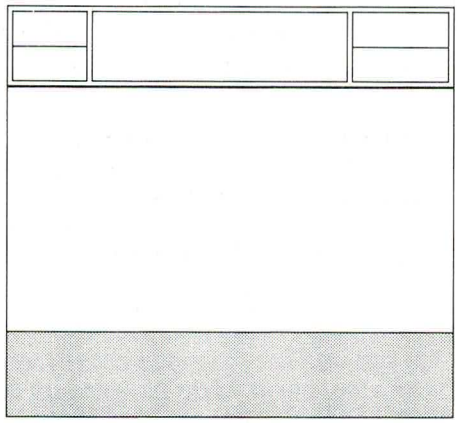

Figure 2.3 The work area

These two areas almost invariably work together, since commands typed into the work area produce their results in the display area.

As an example, type in the following short program, exactly as it is shown below.

```
let x=13:while x>0:print x:let x=x-1:endwhile ENTER
```

The text of this program will appear in the first line of the work area. When you press **ENTER**, the numbers from thirteen down to one will be printed on successive lines of the display area. The bottom line of the display area will be left blank except for a red cursor indicating the next position at which text will be displayed. The numbers from fifteen to one are displayed which, together with the bottom blank line, occupy all sixteen lines of the display area.

The command:

```
cls ENTER
```

will clear the display area completely.

The Control Area

The control area occupies the top few lines of the screen. It shows the normal options: Help (**F1**), to turn the prompts on and off (**F2**), cancel any incomplete operation (**ESC**), and use a command (**F4**).

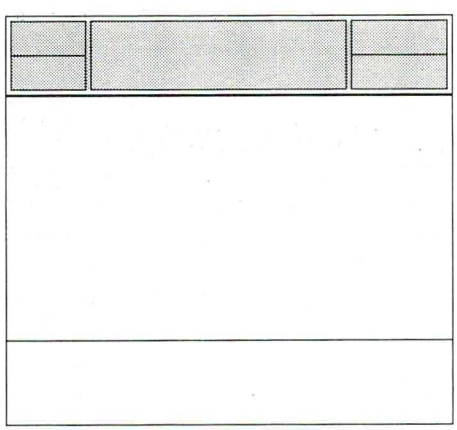

Figure 2.4 The control area

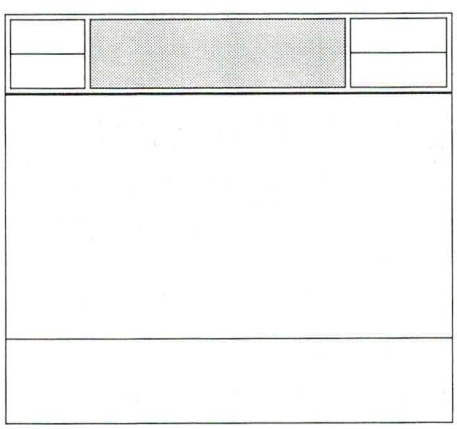

Figure 2.5 The commands

USING THE COMMANDS

Archive's commands form a programming language and you must type their names in full. This may seem long-winded at first, but later you will be shown how to create procedures which allow you to enter commands with a single keystroke.

There are four different lists of commands which can be displayed by pressing **F3**. If a command list is already being shown, pressing **F3** will display the next list in sequence. These commands are used simply by typing in the name and pressing **ENTER**. However, some commands need further information and will ask for it.

You can use any of the commands, even if its name does not appear in the current display in the control area.

THE MODE COMMAND

You can combine the control, display and work areas into a single area with the **mode** command. Used by itself **mode** will combine the three areas into a single area. Typing **mode 0** will also have the same effect. Try

```
mode ENTER
```

and the input from the keyboard and anything displayed by a command or program will share the whole of the screen. A value of 1 divides the screen back into three areas.

You can also use the **mode** command to change the number of characters displayed across the screen. To do this you must supply a second number separated by a comma from the first. The second number must be a 4, 6 or 8 to select a 40, 64 or 80 column display. Try typing

```
mode 0,4 ENTER
```

to change the display to 40 characters and to combine all three areas on the screen. Note that the 0, which originally was optional must be typed to change the size of the display.

Try some different combinations to see the effect on the display. Finish with a command that leaves the screen divided into its three areas, but choose the number of characters that gives a clear display on your television or monitor.

CHAPTER 3
QL ARCHIVE FILES
FILES RECORDS AND FIELDS

An Archive file behaves rather like a card index. A real card index consists of a box containing a set of record cards, each card containing various items of information. For such a card index to be useful, there have to be rules to determine where each piece of information is written.

Suppose, for example, that we have a name and address index. You would normally write the person's name across the top, followed by the address and telephone number (if any). It would be very difficult to use if some cards had the name written at the top and others had it written near the bottom. You would normally expect to be able to use the index by flipping through the cards, reading only the top line, until you found the name you were looking for.

If you had two sets of record cards, such as a set of name and address records and a set of stock records, you would not normally store them both in the same box. You would use two boxes and label them, for example, "Customer Records" and "Stock Records".

The card index system contains most of the ideas necessary to understand how an Archive *file* works. A file is like the card index box and is given a name to identify it. The file is made up of a collection of *records*, each of which serves the same purpose as a record card. A file, then, is simply a collection of related records.

Like a card index, the information in each record is organised in a regular way. Individual items of data, such as telephone numbers might be kept on a specified area of the card. A record in an Archive file is organised in the same way. Each item is stored in a separate region of the record, known as a *field*. A record in a customer file, such as that described above, would contain a name field, an address field, a discount field and so on.

If this were the whole story there would be little point in using an Archive data file in preference to a physical card index. There are, however, many advantages when you use computerised records. A customer record card index would normally be arranged in alphabetical order of customer names which makes it an efficient way to find the information about a particular customer. Suppose, however, you want to send a letter to all your customers who have not placed an order with you during the last six months. It would be a very tedious task to go through the entire contents of a card index to compile such a list. In Archive you can make such a search by using a few simple commands. Furthermore, it is easy to arrange for a set of address labels to be printed at the same time.

You can save a great deal of time and effort by using Archive to store and manipulate your records.

CHAPTER 4
EXAMINING A FILE

The best way to start learning about Archive is to look through the demonstration file **gazet**, provided on the Archive cartridge. This is a file which contains information about various countries – the continent, the capital, the currency, the language, the population, the land area and the gross domestic product per capita.

Most of the examples in chapters 4 and 5 refer to the "gazet" file. Before using it, you should make a copy of it using the following procedure:

When you have loaded Archive, put a formatted cartridge into Microdrive 2 and type:

```
backup ENTER
mdv1_gazet_dbf ENTER
mdv2_gazet_dbf ENTER
```

Wait until the two Microdrives have stopped; be patient as the file is quite long and can take a while to copy. Use the copy, now on the cartridge in Microdrive 2, for experimenting.

From now on we will not always write **ENTER** at the end of every command but please remember that it must still be used.

The **look** command opens a file so that you may read its contents, but you are not able to make alterations or additions to the file. It is a safer command than **open** if you are merely looking through a file because the file is protected against accidental modification. You can examine the copy of the "gazet" file on Microdrive 2 by typing:

```
look "gazet"
```

DISPLAYING A RECORD

To look at the first record type:

```
first
display
```

Don't forget to type **ENTER** after each command and then the display will show the first record of the file.

Note the first line shows the logical name of the file; Archive automatically supplies the name "main" for a single file. Logical file names are usually used when you are using more than one file at a time and are described later.

EXAMINING OTHER RECORDS

Having looked at the first record of the file, you may want to move on to the following record. Type:

```
next
```

and the display shows the next record in the file. When you are typing single commands after a **display** command the display area is continuously updated to show the contents of the current record. You can use the **next** command to step through the file, record by record until you reach the end (it will not pass the last record).

There are three other related commands which you can use to control which record of the file is displayed.

```
back    – which displays the previous record,
first   – which displays the first record,
last    – which shows the last record of the file.
```

Try using these commands to move around the file, displaying any record you like. Note that the four commands **first, last, next** and **back** do not themselves display the record. They merely move from record to record regardless of whether or not you have used **display** command.

SEARCHING A FILE

Find

The first and simplest search command is **find**. This will search from the beginning of a file, looking for the first occurrence of a specified piece of text in any of the text fields.

For example:

```
find "africa"
```

When you press **ENTER** there will be a slight pause and then the first record containing the word 'africa' in any of its text fields will be displayed. Note that this search is independent of whether the text is in upper or lower case and will therefore find 'Africa', 'AFRICA' or 'africa'.

Continue

If the first record that is found containing the text is not the one that you want, you can find the next occurrence by typing:

```
continue
```

The **continue** command will repeat the previous search, looking for the next occurrence of the text in any text field of the following records.

It is possible that you may have to repeat a search several times before finding the record you require. Press **F5** and Archive will put the previous command back in the command line. Press **ENTER** and the command will be executed.

Search

Another method of locating a particular record is to use the **search** command. This allows you to find a record by specifying the contents of one or more specific fields, for example:

```
search continent$="EUROPE" and language$="FRENCH"
```

will find the first record in the file which matches both conditions. You must type in the full field name.

Unlike the **find** command, **search** will only test the fields you specify and will differentiate between upper and lower case letters. Use the **upper( )**or **lower( )** case functions to make the search case independent, for example:

```
search lower(continent$)="europe"
```

Again the **continue** command can be used to find the next occurrence of the text.

Select

In many cases, you may want to look at a sub group of the records within a file. Suppose, for example, you only want to look at the details of countries in Europe. You can use the **select** command to pick out from the file all those records which satisfy a certain condition. The file will then behave as though only those selected records are present. Try this command on the "gazet" file to see how it works. First type:

```
print count()
```

which will tell you how many records there are in the file. Then type:

```
select continent$="EUROPE"
print count()
```

and you will see how many records have been selected. The records that are removed from the file are still held in the computer's memory and you can restore them to the file at any time by using the **reset** command. Type:

```
reset
```

and print the value of **count( )** again, to check that the file has been restored to its original state.

When you use the **print** command from the keyboard, any file shown on the screen will be erased. This is because, in general, **display** and **print** use areas of the screen which overlap. After using **print** you must type **display** again to restore the display.

SORTING A FILE

The file records may not always be in the order you need. You can sort the file by the contents of numeric or text fields. **Only the first eight characters of text are taken into account by order.**

Suppose, for example, you want to sort the records of the "gazet" alphabetically by capital city. You can do this by using the **order** command as follows:

```
order capital$;a
```

The "a" following the semicolon specifies that you want to sort the file in ascending order. Replace it by "d" if you want the file sorted in descending order. The capital$ field becomes the *sort key* for the file. You can specify a sort key composed of up to four fields by

giving a list of fields after the **order** command. For each of the keys you must specify whether the sort is to be in ascending or descending order. The following command, for example, will sort the file into descending order by population and ascending order by capital.

```
order pop;d,capital;a
```

Note that a semicolon separates each field name from the "a" or "d" that specifies ascending or descending order, but that each pair (field name and letter) is separated from the next by a comma.

When more than one field is specified for sorting purposes the records are initially sorted according to the contents of the first field in the list. If two or more records have the same contents for this field, they are ordered according to the next field in the list. If records exist which are equal in respect of the contents of both of these two fields, they are ordered according to the contents of the third field, and so on.

LOCATE

When a file has been sorted, you can use the **locate** command to make any particular record the current record in the file. Its action is to find the first record whose first sort field is greater than or equal to the given expression. This record becomes the current record in the file.

For example, if the "gazet" file has been sorted as described in the last example, the command:

```
locate "100"
```

locates the first country in the sorted file which has a population of 100 million. If there is no such country Archive will locate the first country with a population less than 100 million (remember that the file was sorted in descending order).

Locate is followed by an *expression* which may be either text or numeric, but must be of the same type as the field used to sort the file. (See the *Reference* chapter.)

You can locate a record with respect to the contents of more than one sort field by using **locate** with multiple expressions, separated by commas. For example,

```
let a="100"
let b$="D"
locate a,b$
```

will find the first country with a population of 100 million or less, and with a capital whose name either starts with "D" or is after "D" in the alphabet. In this example Archive will locate Bangladesh, which has a population of 76.1 million and whose capital is Dacca.

The only restriction on the number of expressions that you can use with **locate** is the number of fields used to sort the file.

You cannot use **continue** after **locate**. Repeating a **locate** with the same condition will always locate the same record.

Locate is the fastest way of locating a record in a large, sorted, file. Because of the uncertainty in the record that is located, you may have to make a further check on the record to make sure it is what you want.

CLOSING A FILE

When you have finished looking at a file you must tell Archive. You can do this by typing

```
close
```

This will only act on files and will leave any program or screen layout intact. You can close all your files and clear out your data and display area by typing

```
new
```

This will clear Archive to its initial state after loading.

This only acts on the data files, leaving any program, or screen layout, intact.

Alternatively, if you have finished using Archive, you can go back to SuperBASIC by using **quit**. This command closes all open files automatically before leaving Archive.

Remember that you should never remove a cartridge from a Microdrive while it contains open files.

CHAPTER 5
MODIFYING A FILE

Before typing in examples in this chapter, type **new** first to ensure that Archive is cleared and ready for a fresh start.

The **open** command prepares a file for both reading and writing.

If you open a file with the **open** command, instead of **look** you will be able to write to the file to change its contents as well as read it. This means that any additions, deletions or modifications will make a permanent change to the copy of the file when it is closed. Type:

```
open "gazet"
```

If you have opened a file for reading with **look** then you must not use any commands which will attempt to modify the data. If you do, Archive will report an error. The commands described in this chapter modify data files and so should only be used with a file opened with **open.**

Display the first record of the file with:

```
first
display
```

CLOSING THE FILE

When you have finished modifications to the file you must close the file (using **close** or **new**) to ensure that all the changes are recorded.

If you do not close a file properly (for example, if you just turn off the computer when you have finished) the file may be changed and your most recent changes will not be recorded. **Always make sure that there are no open files on a cartridge before you remove it from the Microdrive. Do not switch off the computer without first closing all open files and removing the cartridges from the Microdrives.**

INSERT

The **insert** command is used to add one or more records to the current file. When you use **insert** you will be asked to type in the contents of each field of the new record. Type:

```
insert
```

The display area will now show:

```
Logical name : main
country$     :
continent$   :
capital$     :
currency$    :
languages$   :
pop          :
area         :
gdp          :
```

You can now type in the contents of each field. You can step from one field to the next by pressing **ENTER** or **TABULATE** or you can step back to the previous field by holding down **SHIFT** and pressing **TABULATE.** You can make as many changes as you like to the fields until you are satisfied. The new record can be inserted into the file by pressing **F5**. Press **F4** to leave **insert**. Try typing:

SCOTLAND	TABULATE
EUROPE	TABULATE
EDINBURGH	TABULATE
POUND STERLING	TABULATE
ENGLISH	TABULATE
10	TABULATE
30	TABULATE
50	TABULATE

The display area should now show:

```
Logical name : main
country$     : SCOTLAND
continent$   : EUROPE
capital$     : EDINBURGH
currency$    : POUND STERLING
languages$   : ENGLISH
pop          : 10
area         : 30
gdp          : 50
```

When you are satisfied that you have typed in the new information correctly, press **F5** to insert the new record into the file. The fields you have just typed in will then be blanked out ready for you to insert a new record. Press **F4** when you have finished inserting.

You can also end the entry for each field and move to the next one by pressing **ENTER**. The new record is added to the file automatically when you press **ENTER** after the last value.

If the file has been sorted the new record is inserted at the correct position to maintain the order.

DELETE

You can use the **delete** command to remove a record from the file. **delete** removes the current record (the one shown by **display**) from the file. All you have to do to remove a particular record is to display it, and, having made certain that it is the correct one, type:

```
delete
```

CHANGING A RECORD

It is also simple to modify the contents of any or all of the fields within an existing record. There are two methods.

Alter

Select the record you want to change (use **display** and **find**) then type **alter**. **Alter** works in the same way as **insert** except each field shows its old contents. You can step over those fields you do not want to change (use **TABULATE** or **ENTER**). Type in a new value or use the cursor keys to modify an old one. Press **F5** to replace the record.

As with **insert**, the record is replaced automatically if you press **ENTER** after the last field in the record.

Update

Select the record you want to change then change the contents of the field variables until the displayed record is as required. Type **update** to change the record.

For example, suppose that you decide that Iceland should be in Europe instead of the Arctic. Find the record by typing

```
find "Iceland"
display
```

Use the **let** command to change the contents of the continent$ field:

```
let continent$ = "Europe"
```

Finally put this change into the record by typing **update**.

In both of the above methods the new record will be inserted in the correct position if the file has been sorted. Otherwise the replacement record is inserted in an unspecified position in the file.

The **alter** command is simpler to use, but always affects the current record. The **update** command can be useful when you are using multiple files.

Remember that you must close the file with the **close**, the **new** or the **quit** command, before switching off the computer.

CHAPTER 6
CREATING A FILE

If you have been following the examples up to this point, you will have been using Archive only to look at the file provided for you. This chapter will show you how to create your own file with your own choice of file names.

If necessary, type **new** to clear anything in the computer's memory and to close any open files. Make sure that the formatted cartridge on which you are going to create the file is in Microdrive 2.

Suppose you want to use ARCHIVE to make a catalogue of your books. To do this, you will have to create a new file called "books". The first thing to do when creating a file is to decide what it is going to contain, that is, what fields you will use in each record. In this case you will obviously need to record the author, title and subject; you may also like to include other details, such as the type (fiction or non fiction), ISBN (International Standard Book Number), shelf location, a brief description and so on. In this example we shall simply use three text fields to contain the author, title and subject and one numeric field which will be used to hold the ISBN.

CREATE

You create a file with the **create** command. You must specify the name of the file to be created and the names of the fields to be used in each record. The $ sign indicates that the field contains text. When you have finished defining the fields of a record you end the **create** command with **endcreate**. You can create a simple book catalogue file, as described above, by typing in the following sequence.

```
create "books"
 author$
 title$
 subject$
 isbn
 endcreate
```

Note that you do not have to type in the final **endcreate** command. You can do so if you want, but you can end the creation of the file simply by pressing **ENTER** on a blank input line. You must, however, include **endcreate** if you use **create** in an Archive program.

ADDING RECORDS

When you have created a file, it is open for both reading and writing, but it contains no records. Records can be added using **insert**. Type:

```
insert
```

and the display area will show:

```
logical name : main
author$ :
title$ :
subject$ :
isbn :
```

All you have to do is to type in the contents of each field. For example, type:

```
Bloggs, J          TABULATE
A Boring Manual    TABULATE
Cannon Making      TABULATE
1234567            TABULATE
```

the display area should show

```
Logical name : main
author$      : Bloggs, J
title$       : A Boring Manual
subject$     : Cannon Making
isbn         : 1234567
```

Insert the record into the "look" file by pressing **F5**. The field value will be cleared ready for inserting another record.

Remember that you can also end the entry for each field and move to the next one by pressing **ENTER** and that pressing **ENTER** after the last value will add the record to the file.

When you have finished press **F1**, and remember to use **close** or **quit** to save the file first.

CHAPTER 7
SCREEN LAYOUTS

When you use the **display** command on a file that you have created, the records are shown using the standard Archive screen layout.

DEFINING A SCREEN LAYOUT

You can design your own screen layout, better suited to the information in your data file. **Open** an existing file and type in:

```
display
```

You select screen editing with the **sedit** command – type in:

```
sedit
```

The display area shows the current screen layout, which will be the one that Archive creates automatically. If there is no screen layout in the computer's memory, the display area may be blank.

You will see that the values of the fields of any file are not included. The spaces where these values are normally shown are marked by rows of dots. You should think of a screen layout as a background against which the values of a number of variables are shown in specific positions. Archive shows a screen layout into two stages – first it draws the background text and then it shows the values of the variables at the marked positions on the screen.

You are initially at the *main level* of the command and you have three options:

type background text into the screen
press **ESC** to leave **sedit**
press **F3** to use a screen editing command

To design a screen layout, press **F3** and then C to clear the screen and make a fresh start. Press **ENTER** to confirm your choice; any other key will return you to the main level of **sedit**.

Choose paper and ink colours by pressing either P or I and pressing any key to switch between the four available colours. Press **ESC** to return to the main level to enter background text.

Background text might be explanatory, such as:

Andrew Young's World Gazeteer

Or it might consist of a new name for one of the fields in your file:

Population (millions):

You can move the cursor to any point in the display area by using the four cursor keys. Anything that you type will immediately appear in the display area at the position of the cursor and will become part of the background of the layout. The only exception is if the cursor is positioned within an area of the screen reserved for the display of a variable. Archive shows the name of the variable in the work area at the bottom of the screen. You cannot type background text into this area unless you first free the area, as described later.

The four screen edit commands enable you to produce attractive and colourful formats for displaying your data. Clearing the screen has already been explained. You may need to experiment to completely master the remaining three so make sure you are using a copy of your data file which is expendable.

SCREEN EDIT COMMANDS

Mark Variable (V)

Suppose you want to show the value of the variable country$ at a particular position in the screen. Move the cursor to that point and press **F3** and then the V key. Archive asks you to type in the name of the variable. You type:

```
country$
```

Note that this name does not appear on the screen – you are just marking the point where the *value* is to be shown. When you press **ENTER** Archive asks you to show

how much space is to be reserved for showing the value. You press any key except **ENTER** to mark the space with a row of dots. **CTRL** and the left cursor key can be used to delete reserved space. When you have reserved enough space you press **ENTER** and Archive takes you back to the main level of **sedit**.

If you move the cursor into one of the reserved areas, (marked by dots), Archive shows the name of the variable for which space is reserved in the work area.

If you reserve space for a variable in a region which overlaps any area that is already reserved, you are given the option of cancelling the old area. You can then use the option again to allocate space for a new variable.

Ink (I) Suppose you want to change the ink colour. Move the cursor to the point where you want the new colour text to start and press **F3** and then the I key. Archive shows the four available colours in the control area. The one that is selected will be the one that is highlighted. Press any key to change the selected colour and then press **ENTER** to record your choice. Any subsequent text that you type will appear in the new colour until the **ink** command is used again.

Paper (P) Changing the paper colour works in the same way – except that you press **F3** and then the P key.

If you want a colour change to affect only part of a line, you should move the cursor to the start of the region and select the paper and ink colours that you require. You should then move the cursor to the end of the region and make a second selection of paper and ink colours, returning them to their original values.

ACTIVATING A SCREEN LAYOUT

Once you have designed a screen layout and have left **sedit**, the screen layout will be *active*. This means that the values of all the variables in the screen layout will be displayed automatically every time Archive completes a command or a program. If, for example, you type the command **next** Archive moves to the next record of the current file and shows those fields that are included in the screen layout. Any active screen is deactivated each time you use the **cls** command.

If a screen layout is not active, you can activate it with the **screen** command. This displays the background text of the screen layout, but does not show the current values of the variables.

SAVING AND LOADING SCREENS

You can save your screen design on a Microdrive cartridge using the **ssave** command:

```
ssave "filename"
```

where "filename" is a name of your choice. The screen layout is saved exactly as it appears.

You can reload the screen layout by typing in the command:

```
sload "filename"
```

When you load a screen layout, it is automatically displayed on the screen and made active.

Archive will not automatically update an active screen layout from within a program. Suppose you want to show all the records of the current file, one after another, and tried to do so by typing the one-line program:

```
first: let x=0: while x<count():next:let x=x+1:endwhile
```

(The **while** and **endwhile** commands cause the section of program that they enclose to be performed repeatedly, while the condition following **while** is true. For correct operation every **while** command must have a matching **endwhile**.)

This program would fail to do what you want, since Archive only updates the contents of the screen layout at the end of the program.

THE SPRINT COMMAND

You can, however, force a display of the values of the variables in an active screen from within a program using the **sprint** command. The following one-line program will show all the records, as required.

```
first:let x=0:while x<count():sprint:next:let x=x+1:endwhile
```

If there is no active screen **sprint** has no effect.

THE DISPLAY COMMAND

Remember that the **display** command uses the standard layout. It will always replace any screen layout with its own simple list of the fields of the current record of the current file. You must therefore **ssave** your screen layout before you next use **display**. If you do not, your screen layout will be replaced and you will not be able to get it back again except by redesigning it with **sedit**.

CHAPTER 8
PROCEDURES

To use the examples in this chapter, first type **new** to clear the computer, then type **look "gazet"** to open the example file on your data cartridge, which is assumed to be in Microdrive 2.

The commands and functions of Archive together form a programming language which you can use to write programs that will manipulate your files. You will find that Archive programs are simple to write.

An Archive program is made up of one or more separate sections. Each section is known as a *procedure* which is simply a named section of program. You can refer to a procedure by its name, like the procedures which you write and use in SuperBASIC. In Archive you can run a procedure by typing its name at the keyboard. When you write a procedure you are effectively adding a new command to Archive.

No procedure may contain more than 255 lines, and each line must not contain more than 160 characters.

CREATING A PROCEDURE

You use the *program editor* whenever you want to write or change a procedure. This editor allows you to change, delete or add to the text of procedures.

The program editor is described in detail in Chapter 9, but in this chapter we will look briefly at some of its features so that we can write a few short procedures. We shall assume that initially there are no procedures in the computer's memory.

Type:

```
edit
```

to enter the program editor. The control area changes, showing that you should type in the name of the procedure. Entering the editor will always allow you to create a new procedure if none are defined or loaded.

The first thing to do, therefore, is to decide what the new procedure should do. Let us start with a very simple task; to make life easier by renaming the **display** command. We will save typing by giving it the name "d".

Just type

```
d
```

The left hand side of the display area now shows the name, and the right hand side a listing of the procedure. The procedure, as yet, contains no commands; the **proc** and **endproc** which mark the beginning and end of the procedure were automatically added by Archive.

The *body* of the procedure must be added; that is sequence of actions it is to perform.

The control area shows that you can add lines of text to the new procedure. In terms of the current example this text is the **display** command. Type:

```
display
```

and Archive will insert the new text into the procedure below the highlighted line. If you have followed this example the display will contain:

```
d   proc d
    display
    endproc
```

You could add more lines of text – each line would be inserted below the highlighted line.

In this case, however, the procedure is complete so you can leave the **edit** command by pressing **ESC** twice.

All you have to do to use the procedure is type its name, followed by **ENTER**. This new procedure will perform the same function as typing the command **display** in full.

LISTING AND PRINTING PROCEDURES

Whenever you call the **edit** command you are shown a list of the names of all the defined procedures present in the computer's memory.

You can list any one of these procedures from within **edit** by pressing the **TABULATE** key to move down the list or the **SHIFT** and **TABULATE** keys together to move up the list until the particular procedure name is highlighted. The procedure is automatically listed at the right hand side of the screen. If the procedure is too long to fit in the display area, you will be shown the first part and you can then scroll up and down through the procedure using the up and down cursor keys. When you have finished you can leave the **edit** command by pressing **ESC**.

If you want a printed listing of your procedures you can use the **llist** command. Type:

```
llist
```

and all the procedures currently in the computer's memory will be listed on a printer.

WARNING: Do not use this command unless a printer is attached since this will cause the program to "hang".

SAVING AND LOADING PROCEDURES

If you want to keep the procedures that you have defined, you can use the **save** command. This stores all defined procedures in a single named file on Microdrive cartridge. If you want to save the new display procedures that you have just defined in a file called "myprocs", you should type in

```
save "myprocs"
```

At any later time you can bring these procedures back into the computer's memory by typing:

```
load "myprocs"
```

The **load** command deletes any existing procedures in memory before loading the new ones from the Microdrive cartridge. If you want to add the new procedures to those already in memory, you can use the **merge** command. For example:

```
merge "myprocs"
```

This works like **load**, except that the existing procedures are not deleted. If a new procedure has the same name as an existing one, the new one will replace the old version.

EXAMINING FILE RECORDS

Renaming commonly used commands with single-character names is one way of making life easier for yourself. An alternative would be to write a longer procedure to replace several commands by single key presses. Try using the **edit** command to define the following procedure. It allows you to open and examine any of your data files, providing, of course, that the file you wish to use is not already loaded.

If you have already defined a procedure, typing:

```
edit
```

will not automatically give you the option to create a new procedure. From within **edit** you must press **F3** and then the N key to start a new procedure.

Don't worry if you make a few mistakes while typing in the example – you will learn how to correct them in the next chapter.

```
proc vufile
  cls
  input "which file? ";file$
  look file$
  display
  let key$="z"
  while key$<>"q"
    sprint
    let key$=lower(getkey())
    if key$="f":first:endif
    if key$="l":last:endif
    if key$="n":next:endif
    if key$="b":back:endif
    endwhile
  close
  endproc
```

Remember that you leave **edit** by pressing **ESC** twice.

You can use the procedure by typing:

```
vufile
```

It will first clear the display area and then prompt you to type in a file name such as "gazet". If "gazet" is already loaded, however, you will receive an error message. To recover, type **new** and load and run the procedure again. When you have entered the name of one of your data files the procedure will open that file in read-only mode and display its first record. It will then wait for you to press a key and will respond to the keys f, l, n, b or q. The first four of these will cause the appropriate display action (first, last, next or back) and pressing the q (quit) key will close the file and end the procedure.

Since this is the first program of any great length that we have written, a few comments might prove helpful. First note how the example is indented to clarify the structure of the procedure. There is no need for you to type it like this, the indents are added automatically as you write, list or print the procedure.

The main part of the procedure (waiting for a key to be pressed and performing the appropriate action) is enclosed between **while** and **endwhile** commands. This repetitive loop will only be left when the condition following **while** is false, in this case, when you press the q key.

The **if** command, used several times within this loop, also requires that each **if** has a matching **endif** to mark the end of the sequence of instructions to be executed if the condition is true. **If** and **endif** are separate commands and can be used on different lines. We could, for example, have written the first of the **if** statements in this procedure as:

```
if key$="f"
  first
  endif
```

You may include several lines of statements between **if** and **endif**; they will all be executed, provided the condition following **if** is true. In the **vufile** procedure these statements are sufficiently short that each can be written on a single line, using the colon to separate the individual statements.

As you can see, a **sprint** command is used within the main loop of this procedure to make sure that each new record is shown on the screen. Remember that, although the display commands (**first**, **last** etc.) always move to the correct record, the data in the display area is not automatically changed until the end of the procedure. If we had not included the **sprint** command, no information would have been shown in the display area until you pressed the q key to leave the procedure. In that case all you would see would be the result of the last of any sequence of keypresses that you have made.

CHAPTER 9
EDITING

This chapter describes the program editor. We shall include a few simple examples, but the best way to learn is by using them yourself. Start by typing **new** to clear the computer's memory.

When you have read this chapter you could try writing a few simple programs of your own, or you could try modifying the procedures you typed in while working on the last chapter. If you want to use longer examples you could use the editor to type in all or part of the programs in the following chapters.

THE PROGRAM EDITOR

You enter the *main level* of the program editor with the **edit**

As an example we can create a procedure and add a couple of statements to it. From the main level of **edit**, press **F3** and N to create a new procedure. Type in **test** when prompted for the name of the procedure.

Press **ESC** twice to leave the editor without adding any statements. Then use the **edit** command again. If you have no other procedures loaded, the screen will show:

```
test   proc test
          endproc
```

If the procedures you created in the last chapter are still loaded, then **test** is highlighted on the left as the current procedure among these other procedures. Press **F4** to insert lines of text. The line containing **proc** will be highlighted.

Now type:

```
print "this is a test" ENTER
print "there are two statements" ENTER ENTER
```

Pressing **ENTER** twice in succession takes you out of **insert**. When you have finished the screen will look like:

```
test   proc test
          print "this is a test"
          print "there are two statements"
          endproc
```

The line containing the second print statement is highlighted.

Remember that until you press **ENTER** you can use the line editor to correct any text that you type. However, once you have pressed **ENTER** the line is inserted into the procedure. To get it out again to **edit** it you must press **F5**. Pressing **ENTER** will then replace the old line with the new line.

You are not allowed to edit the **endproc** statement at the end of the procedure. You are also not allowed to edit the word **proc** but you may edit the rest of the contents of this line. You can, therefore, rename a procedure by using the line editor to delete the old name and replace it with a new one. The list of procedures at the left of the screen is rearranged automatically to keep the procedures in alphabetical order.

Editing Commands

There are four separate editing commands which you will have noticed in the command section when creating a new procedure. You can select one by pressing **F3** and then typing the first letter of its name.

New Procedure (N)

You type in the name of the procedure you want to create. If you type in the name of an existing procedure, you will not be allowed to create a second procedure but will be offered the option of editing the existing procedure.

When you press **ENTER** at the end of the name the new procedure becomes the current one, listed at the right of the screen. You are presented with an empty procedure – that is, one containing only the **proc** and **endproc** statements.

Delete Procedure (D)

This command deletes the current procedure from your program. You must first select the procedure you want to delete by using the **SHIFT** and **TABULATE** keys, as described earlier, to make it the current procedure. You then select the command by pressing **F3** and then the D key.

You must press **ENTER** to confirm that you really do want to delete the procedure. If you change your mind at this stage you can press any other key to go back to **edit** without deleting the procedure.

Be careful when you use this command since there is no way to restore a deleted procedure, except by typing it in again.

Cut (C)

This command removes one or more lines of text from the current procedure. The text that is removed can be inserted in another position, or even in another procedure, by means of the **paste** command.

Before you select the command you should use the up and down cursor keys to make the current line either the first or the last line of the section you want to remove. You can then select the command by pressing **F3** and then the C key.

If you then press **ENTER** the current line will be removed from the procedure. Alternatively you can use the up or the down cursor key to move the cursor to the other end of a section of text that you want to remove. The region of text that will be removed is marked by highlighting. When you have marked the text you want to remove you should press **ENTER**. Archive will immediately remove the marked text.

Paste (P)

This command inserts the text removed by the last use of the **cut** command into the current procedure, below the current line. The text can be inserted in another position, or even in another procedure.

Before you select the command you should, if necessary, use the **SHIFT** and **TABULATE** keys to select the procedure in which you want to insert the text. You should also use the up and down cursor keys to highlight the line immediately above the position where you want to insert the text.

Archive immediately inserts the text, underneath the current line. When you have used **paste** to insert the text, the paste buffer is empty. You can not, therefore, insert the same text in more than one position.

CHAPTER 10
PROGRAMMING IN ARCHIVE

This chapter will describe the development of an actual working example and each new technique will be described as it is needed.

Suppose you are involved in running a club or society which charges a subscription and produces a newsletter. You will need to send a copy of each issue to every paid-up member. You will also need to send a reminder to each member when his or her subscription falls due.

This example allows you to construct a mailing list and then print a set of address labels on request. The address label includes a reminder when a subscription is due. The example assumes that you send out six issues of the newsletter per year and that a person's subscription falls due when he or she has received six issues. It could easily be adapted to any situation where you regularly send out some form of circular letter to a number of people on a mailing list.

A MAILING LIST

In this example we shall make as much use as possible of the existing facilities and introduce some new ones. If you need help with a feature or command you have not yet encountered, or one that seems to do things you don't understand, you may now find it quicker to look for help in the reference section or use the **help** function by pressing **F1**. We use the **insert** and **alter** commands for all additions and changes to the file records. We shall, however, need to write special routines to print out the address labels.

We shall have to cater for the following set of requirements:

Add a new record to the file.
Delete a record.
Modify a record.
Record subscription payments.
Produce the address labels.
Leave the program.

We shall write a procedure to handle each of these tasks and link them together by another procedure which will allow you to select any of these options.

In this application it is quite clear what fields each record must contain. The name and address are essential plus one field to record the number of issues the person has received. We can create the necessary file immediately, as shown below.

```
create "mail"
  title$
  fname$
  surname$
  street$
  town$
  county$
  postcode$
  issues
  endcreate
```

We have used three string fields for the person's name; to hold the title (Dr, Mr, Mrs etc.), the first name and the surname respectively. We could probably have managed with just a single field.

There are four string fields for the address, nominally reserved for the street address, the town, county and postcode. You do not always have to use them in this way, but can treat them as four general fields to hold the address. Four fields should normally be quite sufficient.

There is only one numeric field, to hold the information about how many issues remain to be sent.

Now that we have the file, we can use it to test the various procedures as we write them. It is a good idea to test each procedure as far as possible as you go along. You can then spot each mistake as it occurs and correct it immediately. If you leave all the testing to the end it will be much more complicated as several things may be going wrong at the same time. Keep things as simple as possible while you are still testing your procedures. Try to make sure that each procedure works correctly before you move on to the next one. That way you will find that your final program will usually work as soon as you have written the last procedure.

Insertion

We do not need to write a procedure to add a record. We can use **insert**. Remember that you must use **sprint** to force the display of the contents of the record from within a procedure. You can use **insert** immediately to add a few records to the file so that you can test the other procedures on a real file.

Deletions

At some time you will want to remove the records of people who have not renewed their subscriptions. We shall write a procedure, **wipe**, which allows you to scan through the file, examining the records of all people who have not renewed, and to decide which should be deleted.

We shall use the field variable **issues** to hold the number of issues that a person is entitled to receive. All records for which the value of **issues** is zero are therefore candidates for deletion.

```
proc wipe
  rem ***** delete non-paying subscribers *****
  cls
  display
  select issues =0
  all
    sprint
    print at 10,0; "DELETE (y/n)? ";
    let ok$ =lower(getkey())
    print ok$
    if ok$ ="y"
      delete
      print "DELETED"; tab 15
      else
      print tab 15
      endif
    endall
  reset
  endproc
```

Since a deleted record cannot be recovered, the full contents of the record are displayed and you are asked to confirm that you really want to delete it. We use the **getkey()** function which waits for a key to be pressed and then returns the ASCII code of that key. Note that **lower()** converts the code to the lower case character so that you can type the letter in either upper or lower case.

Once you are satisfied you have correctly entered this procedure, you may try it out on your file, (provided, of course, that you have entered some test records). First, leave **edit** by pressing **ESC** (twice if necessary) and save your procedure in a file called "Maillist".

Type:

```
save "Maillist"
```

The procedure called **wipe** is now stored and can be called whenever "Maillist" is loaded.

After entering each of the following procedures, repeat these steps, each time storing the new procedure in "Maillist".

Payments

You will normally want to record a batch of subscription payments from a list of names and addresses. You will therefore need to get the record of a particular person. The quickest way is to write a separate procedure, **getrec**, to locate a particular record and then incorporate it in a **pay** procedure.

The **getrec** procedure asks for a text string (n$) and then locates the first record in the file which contains that text. If you reply by just pressing **ENTER**, n$ is set to the empty string and no search is made. This will, however, indicate that you have finished recording payments.

From the **edit** level, press **F3** and N to start entering **getrec**.

```
proc getrec
  rem ***** locate a particular record *****
  cls
  let ok$ ="n"
  input "who? "; n$
  if n$ <>""
    find n$
    while ok$ <>"y" and found()
      print title$ ; " "; fname$(1); " "; surname$
      print street$
      print "OK (y/n)? ";
      let ok$ =lower(getkey())
      cls
      if ok$ <>"y"
        continue
        endif
      endwhile
    if not found()
      print n$ ; " not found"
      endif
    endif
  endproc
```

The search uses the **find** command, so that the text is found in any string field. You can therefore identify a record by name or by address. Of course, the first record which matches may not be the one you want, so we have to be able to continue the search. This is the purpose of the **while endwhile** loop. This prints out the name and first line of the address, to identify the record, and asks you if that is the right record. If you do not respond by pressing the Y key, it continues the search. The loop ends either when you answer by pressing the Y key or when the text is not found in any of the remaining records. Note that the function **found()** returns a true (non-zero) value if the search is successful.

Since ok$ could initially be "y" (from a previous successful search) we must give it some other value at the beginning of the procedure, before entering the loop. This makes sure that the loop will be used at least once.

We can now write the **pay** procedure:

```
proc pay
  rem ***** record subscription payment *****
  cls
  let n$ ="x"
  while n$ <>""
    getrec
    if ok$ ="y"
      let issues =issues +6
      update
      endif
    endwhile
  endproc
```

The loop in this procedure continues until n$ is an empty string. This allows you to record several payments without having to select the **pay** option for each one. When you have finished, just press **ENTER** in response to the "who?" prompt. If the value of ok$ is "y" after the call to **getrec** then the payment is recorded by marking it as valid for a further six issues.

Again we have to set the initial value of n$ to some appropriate value (anything except the empty string) to make sure that the procedure is not affected by a previous operation.

Changes

The procedure to allow you to change the contents of a record is now very easy. Again you must be able to select a particular record to change, so the general structure can be identical to **pay**.

```
proc change
  rem ***** alter record *****
  let n$ ="x"
  cls
  while n$ <>""
    getrec
    if ok$ ="y"
      alter
      cls
      endif
    endwhile
  endproc
```

PARAMETERS

We shall now take a short break from the development of the program to describe the use of *parameters* with procedures. You can use a *parameter* to pass a value to a procedure, rather than using the value of a variable. We shall show you a few examples of how they can be used. You do not need to save these procedures in "maillist" and you may delete them before moving on to the section of the program which deals with labels.

Try the following simple example. Using the line editor, you add the parameter to the line containing the procedure name.

```
proc test; a
  print 5*a
endproc
```

This defines a procedure called **test** which requires one parameter, "a". Notice that the parameter is separated from the name of the procedure by a semicolon. Whenever you use the procedure you must always supply a value for the parameter. For example, you could type:

```
test; 3
```

which will print the value 15 – the number (3) has been passed to the procedure as the value of the variable a.

You may specify any number of parameters for a procedure, provided you separate them by commas. For example:

```
proc trial; a,b,c
  print a * b * c
endproc
```

which you can call by:

```
trial; 3,4,5
```

The values you supply do not have to be literal values, but could be variables, as shown below:

```
let x = 2
let y = 5
let z = 7
trial; x,y,z
```

Note that the names of the variables do not have to be the same as the names used within the procedure. We can distinguish between the *formal parameters* (e.g. a,b,c) in the definition of the procedure, and the *actual parameters* which are the actual values that are passed to the procedure.

You can also pass the results of expressions:

```
trial; x*2,z/y,(z-y)*x
```

You are not restricted to using numeric variables but can also pass strings (or string expressions) as parameters, provided you specify string variables in the definition of the procedure. For example:

```
proc try; a$
  print a$
endproc

let t$ = "message"
try; t$
```

The only requirement is that the number and types of parameters supplied must match the list of formal parameters in the definition of the procedure.

Address Labels

The reason for the brief interlude about parameters is that they give a neat way of writing the procedure to print an address label. For the purposes of testing we shall first write the procedure to show the addresses on the display and later convert it to send the output to the printer. We shall assume that the labels are eight lines of print-out in length. If this is not right for your printer and label combination you will have to change the number of lines of space in the procedure so that it matches your requirement. Remember to start saving your procedures in "Maillist" again.

First we shall write a procedure that displays a single line, the contents of which are passed via a parameter.

```
proc doline; x$
  print x$
  endproc
```

We can now use this procedure to display eight lines of text for the address label.

```
proc dolabel
  rem ***** print labels *****
  if issues
    if issues =1
      doline; "REMINDER - Subscription Now Due"
      else
      doline; ""
      endif
    doline; ""
    doline; title$ +" "+fname$ (1)+". "+surname$
    doline; street$
    doline; town$
    doline; county$
    doline; postcode$
    doline; ""
    let issues =issues - 1
    update
    endif
  endproc
```

The procedure includes a reminder in the address label if the person is about to receive his or her last issue. Each time a label is printed, that person's issue count is reduced by one. If this number has reached zero then the label is not printed.

You can begin to see how useful parameters can be – without them this procedure would be much longer. Look how easy it is to combine the title, initial and surname for the first line of the address.

Perhaps you are wondering why we went to the trouble of defining **doline** when we could have just used **print** statements throughout **dolabel**. The reason is that the routine in its present form shows the addresses on the display screen. We can convert it to send its output to the printer merely by changing one line in **doline**, instead of having to change every print statement in **dolabel**. All we need to do is change **doline** to read:

```
proc doline; x$
  lprint x$
  endproc
```

Finally we can write the procedure to print all the address labels:

```
proc despatch
  cls
  all
    dolabel
    endall
  endproc
```

Leaving the Program

The final option is to leave the program when you have finished. This procedure can be very simple – all it has to do is to make sure that the file is closed properly before returning control to the keyboard. We have also added a short sign-off message to make it clear that the program has ended.

```
proc bye
  close
  print "bye"
  stop endproc
```

ERRORS

It is quite likely that sooner or later you will make an error while using this program. You may, for example, accidentally press the **ESC** key or you may type in some text when a number is expected. This type of mistake is detected by Archive and normally results in the display of an error message and a return from your program to the keyboard.

You can use the **error** command to mark a procedure to be treated specially if any error is detected. Any error occurring in the marked procedure, or any procedure that it calls, results in an immediate, premature, return.

The normal method of handling errors is *switched off* for the marked procedure and it is left to you to decide how to deal with it. You can find out the number of the last error that occurred by using the **errnum( )** function. You can use it to read the error number more than once as the value is only cleared to zero by the next use of the **error** command. If no errors have occurred since the start of the program, or since the last time **error** was executed, then **errnum( )** will return a value of zero.

This method, although not easy to understand at first, gives you a very powerful and flexible control of how to deal with errors. The following example shows a typical way of using **error**. It gives you an error-resistant method of inputting a number.

```
proc dotest
  input x
  endproc

proc test
  let n =1
  while n
    error dotest
    let n =errnum()
    if n
      print "You made error number " ;n ;", try again"
      endif
    endwhile
  endproc
```

The first procedure simply waits for your input to the variable x. The second procedure handles any error during the execution of the input procedure. If any error occurs within **dotest** it will be terminated prematurely and the error number will be set. This number is then read by **errnum( )** and, if it is non-zero, the error message is printed (this error message could, of course, be anything you like). Since these statements are enclosed in a **while endwhile** loop, any error will cause them to be executed again. The error number is cleared by **error**, ready for the next try. You can not leave **test** until you have typed in a valid number.

This example reports the number of the error that was detected. On most occasions you will not be concerned about which error occurred. The main use of **errnum( )** is to differentiate between there being no error and there being a detected error of any type. A list of error numbers and possible explanations is included in the *Reference* chapter.

We can now write a procedure which will allow you to select any one of the six options with a single keypress. It is sufficiently simple that no explanation is necessary.

```
proc choose
  rem ***** choose an option *****
  cls
  print
  print " Add Despatch Pay Change Wipe Quit";
  print "? ";
  let choice$ =lower(getkey())
  print choice$
  if choice$ ="a": insert : endif
  if choice$ ="d": despatch : endif
  if choice$ ="p": pay : endif
  if choice$ ="c": change : endif
  if choice$ ="w": wipe : endif
  if choice$ ="q": bye : endif
  endproc
```

All that remains to be done to complete our program is to write a start-up procedure which opens the file and calls **choose**. We must include **choose** in a loop so that you are offered the options again, each time you complete your previous selection.

You will see that the **while endwhile** loop in the following procedure will never end. Such a loop will only come to an end when the expression following **while** has a zero value. In the above procedure the expression always has the value 1, so the loop will continue indefinitely. The only way of leaving this loop is to choose the Quit option. The **stop** command in **bye** immediately returns control to the keyboard.

```
proc start
  ***** rem start procedure *****
  cls
  open "newmail.dbf"
  while 1
    error choose
    let n =errnum()
    if n
      print "Mistake - Press any key to continue"
      let m$ =getkey()
    endif
  endwhile
endproc
```

Within this loop is a sequence of statements which handles any errors, using a similar method to that described in the previous section. If you make a mistake the program will not continue until you press a key. This allows you to look at what you have just done so that you can find out how you made the error.

THE RUN COMMAND

The main procedure in the mailing list program is called "start". This is so that you can use the **run** command when using the program. We have already used this command when we used the "loader" program to load the "gazet" data file

Save this final procedure in "maillist". When you want to run the program you will need to load the procedures into the computer's memory and then execute the main procedure, which will call all the others. One way is to use the **load** command and then type in the name of the main procedure, for example:

```
load "maillist"
start
```

The **run** command will load a named program and then automatically execute the procedure called "start" (if it exists). You can run the program exactly as in the previous example just by typing:

```
run "maillist"
```

The remaining two sections of this chapter include some general purpose procedures which you may find useful.

LOCAL VARIABLES

Most variables that appear in procedures are *global*. This means that they are recognised throughout the program. They may be used or changed in any procedure, and not just the procedure in which they are first assigned a value.

The variables used as formal parameters in a procedure are *local variables* and they are not recognised outside the procedure in which they appear.

The following example may help to make the distinction clear. Before going on, type **new** to clear the computer's memory. First we create a procedure which uses two local variables a and b$, as well as assigning values to two normal (global) variables u and v$.

```
proc demo; a,b$
  print a,b$
  let u=3
  let v$="text"
  print u;v$
  endproc
```

Then we use **demo**:

```
demo 5;"words"
```

All four values are printed showing that all four variables are recognised inside **demo**. Typing

```
print u;v$
```

shows that both of these variables are also recognised outside the procedure. However, typing

```
print a,b$
```

results in an error because a and b$ are not recognised outside **demo**. All formal parameters are local variables, but you can also declare other variables to be local, as in the following example:

```
proc dumbo
  print "inside dumbo"
  print p; q; r
  endproc
proc dummy
  local q,r
  let p = 2
  let q = 3
  let r = 4
  print "inside dummy"
  print p; q; r
  dumbo
  endproc
```

If you attempt to use **dummy** by typing:

```
dummy
```

you will find that the values of p, q and r are all recognised (and therefore printed) in **dummy**, but **dumbo** does not know the values of q and r, which are local to **dummy**.

The values of local variables are not defined anywhere except in the procedure in which they are declared – not even in procedures called from the declaring procedure. The variable p is global and is recognised everywhere.

You may be wondering why local variables are necessary. To illustrate their usefulness, suppose you write a program containing several procedures that you, or someone else, originally write for use in other programs. It is quite possible that two or more of these procedures might use variables with the same name for quite different purposes. If these variables were global then one procedure could alter a value so that it would be wrong for another. In such a situation you would have to check all the procedures that you use and, if necessary, change the names of the variables. If, however, the variables were local it would not matter if they had the same name. Provided they were in different procedures, changing one would have no effect on the other.

Furthermore, it does not matter if a procedure calls another which uses the same name for a variable – provided at least one of them is local. For example, the procedure **choose** in the section on errors, earlier in this chapter, declared the variable choice$ to be local. This means that there is no need to check whether any of the many procedures called by **choose** also use choice$ – the called procedures cannot change the value of choice$ in **choose**.

PROMPTS

Displaying a prompt and waiting for a key to be pressed is one of the most commonly needed actions, so it is worth writing a general-purpose procedure. The procedure must be able to display a wide range of messages. A simple way of allowing the procedure to print any message is to pass the message to the procedure in the form of a parameter.

```
proc prompt; m$
  print m$ + ": ";
  let x$ =lower(getkey())
  print x$
  endproc
```

The message to be displayed is passed to the procedure as a parameter in the local variable m$. The function **getkey( )** waits for a key to be pressed and returns the ASCII code for the key. In this procedure the ASCII code is converted to lower case by the function **lower( )**, so that the result is independent of upper or lower case. Finally the resulting value is assigned to the variable x$. This is a global variable, so that the key that was actually pressed is available to any other procedure in the program.

PAUSE

A useful procedure is **pause**. It uses **prompt** to print a message and then simply waits until a key is pressed. Since you are not usually interested in knowing which key was actually pressed, it uses a local variable, y$, to preserve the original contents of x$.

```
proc pause
  rem ***** wait for any key *****
  local y$
  let y$ =x$
  print
  prompt; "press any key to continue"
  let x$ =y$
  endproc
```

DATA ENTRY

Text

Accepting text as typed input is quite simple. Any collection of characters is a valid text string (even if it does not make sense) and will not cause an system error. You will not normally need to take any special precautions when accepting text input. It will usually be sufficient to use a line such as the following, which asks you to type in your name:

```
input "Please type your name: ";name$
```

Note that a space is included as the last character of the prompt text; this small point makes a lot of difference to the appearance of your program when you use it.

You can input several items with one input statement. All you have to do is to include all the prompts and variable names, separated by semicolons.

```
input "Your first name? ";fname$;"Your surname? ";sname$;
```

This last input statement also ends with a semicolon – this stops the cursor moving to the following line after you have typed your input.

Numbers

When you use the **input** command to enter text to a string variable the computer will accept anything that you type, without complaint. If, however, you try the same thing with input to a numeric variable you will get an error message if you type anything except a valid number. Assuming that you do not want to leave your program every time your finger slips while you are typing in a number, you must make sure that your program can cope with such errors.

The most useful way is to make use of the **error** command, which was described earlier. The following procedure, for example, will accept any valid number within a specified range. It even provides the display of any prompt message you want to appear.

```
proc getnum; m$,min,max
  rem ***** get number in range *****
  local wrong
  let wrong=1
  while wrong
    print m$; "? ";
    error readnum
    let wrong=errnum()
    if not wrong
      if num<min or num>max
      let wrong=1
      print "Allowed range is ";min;" to ";max
      endif
    endif
    if wrong
      print "Try again"
      endif
    endwhile
  endproc
```

Since **error** must be followed by the name of a procedure, we define **readnum** to input a value for the variable **num**.

```
proc readnum
  input num
  endproc
```

Suppose you want a procedure that checks that a number is within the range 1 to 10. You can do this using **getnum** in the following way:

```
proc check
  getnum; "Numeric value?",1,10
endproc
```

CHAPTER 11
USING MULTIPLE FILES

LOGICAL FILE NAMES

This chapter extends the explanation of how to use the Archive programming language by describing how to work with two or more open files. When you have more than one file open at the same time you must be able to identify which file you want to use for any particular operation. You must give each file a unique *logical file name* when you open or create it and then refer to it by that name in all commands that refer to the file.

Archive automatically supplies the *logical file name*, "main", when you open a single file. It is called a logical file name to distinguish it from the *physical file name* – the name you give to the file when you save it.

Since a program refers to a file by its logical file name, you can write a program that will work with several different files.

Logical file names are essential for multiple file operations since you can only open a second file by using both its physical file name and its logical file name. Note that the logical file name is not saved with the file when it is closed and must be specified each time the file is opened.

Two or more data files could contain fields with the same name. When this happens you can identify the file to which the field belongs by adding the logical file name to the field name. For example, if the field country$ appears in two files whose logical file names are "main" and "b" you could refer to each of them respectively as "main.country$" and "b.country$".

CHANGING THE RECORDS OF A FILE

The first example demonstrates how to add, delete or rename fields within an existing file.

Suppose that you want to make some changes to the "gazet" file, to create a new file containing only European countries. The "continent$" field becomes irrelevant and we need not include it. We shall also rename the "pop" field as "population".

The most convenient way of changing the file is to create a second file containing the fields you want and then to copy the required records from the old file to the new one. Let us call the new file "europe". The following procedure will do the rest of the work.

```
proc start
   rem ***** create europe file *****
   create "europe" logical "e"
      country$
      capital$
      languages$
      currency$
      population
      gdp
      area
      endcreate
   look "gazet" logical "g"
   select continent$="EUROPE"
   all "g"
      print at 0,0;g.country$;tab 30
      let e.country$=g.country$
      let e.capital$=g.capital$
      let e.language$=g.language$
      let e.currency$=g.currency$
      let e.population=g.pop
      let e.gdp=g.gdp
      let e.area=g.area
      append "e"
      endall
   close "e"
   close "g"
   print
   print "DONE"
   endproc
```

THE CURRENT FILE

You can see, from this example, that you can use the same name for a field in both files – they can be distinguished by including the logical file name. If you do not include the logical file name then it will be assumed that the *current file* is to be used. The last file to be opened automatically becomes the current file. In this example the current file will be "gazet" (with logical file name "g") so we could make use of this by simply writing the g before the field name in the previous program.

If you do not include the logigal file name in any case where it is optional, Archive will assume that the command refers to the current file. It is usually safer to include the logical file name explicitly, to avoid any possibility of confusion.

You can, at any time, specify the current file by means of the **use** command. If you included the command:

```
use "e"
```

in the above example, then "europe" would be the current file until you changed it again, either by opening another file or by means of the **use** command.

STOCK CONTROL

Now for a more complex example. In a stock control system you will need to:

Find information on a particular stock item.
Obtain a report on the current stock levels of all items.
Record sales and modify the stock records accordingly.
Order new supplies, to maintain adequate stock levels.
Record deliveries of stock.

You will obviously need a file to hold the details of all items held in stock and it is convenient to have a second file to hold details of all your suppliers. You will need to be able to access either file from the other – for example you may want to know all the possible suppliers of a particular item, or to find out what items are supplied by a particular company.

In order to keep the application as simple as possible we shall not use the menu-driven approach of the examples in the previous two chapters. We shall write it as a series of separate commands which can be used – like the standard commands – by typing their names.

Since the procedures will be strongly dependent on the file structure we use, we must first give some thought to their appearance.

The Stock File

The stock file must contain full details of the stock situation for each item. The following list explains all the fields we shall use.

Field Name	Use	Example
`stockno$`	The internal stock code	`A101`
`description$`	Item description	`Widget, large`
`qty`	Number in stock	`500`
`sellpr`	Selling price	`1.25`
`reorderlev`	Reorder when stock level falls below this value.	`200`
`buyqty`	How many to order	`400`

We can create the file by:

```
create "stock" logical "sto"
  stockno$
  description$
  qty
  reorderlev
  sellpr
  buyqty
  endcreate
```

The Supplier File

This file holds the names, addresses and telephone numbers of the companies that supply the goods you sell. It will be useful also to include the name of a contact person in the company. In order to be able to access this information efficiently we shall include a code for each company. We shall use the following fields:

Field Name	Use	Example
coname$	The company's name	Wonder Widgets plc
street$	First line of address	27 Belmont House
town$	Second line of address	LIVERPOOL
county$	Third line of address	Merseyside
postcode$	Last line of address	L31 2HK
contact$	Name of a contact	Andrew Cummins
tel$	Telephone number	051-532 7133
code$	Your code for the company	a

We can create the file by:

```
create "supplier" logical "sup"
  coname$
  street$
  town$
  county$
  postcode$
  contact$
  tel$
  code$
  endcreate
```

The Orders File

This file forms the link between the previous two files. It uses the following fields:

Field Name	Use	Example
stockno$	Your stock code	A101
code$	Your code for the supplier	a
scode$	The supplier's code for the item	123-456
price	The supplier's selling price	0.87
delivery	The supplier's delivery time, in days	28

Each record in this file links one record in the stock file with one record in the supplier file. The above example shows that Wonder Widgets (supplier code "a") can supply you with large widgets (stock code "A101"). In addition, we include details of the price, delivery time and the supplier's own stock code. These items are useful when you order more stock.

Using this file allows you to cater for the cases where one supplier supplies more than one stock item (equal values for code$, but different values for stockno$) and where one stock item is obtainable from several suppliers (equal stockno$ but different code$).

Create the file with:

```
create "orders" logical "ord"
  stockno$
  code$
  scode$
  price
  delivery
  endcreate
```

Enquiries

Having created these files, we now need some procedures to handle the information they will contain. You will find that the most frequently-needed facility is to find information about a particular stock item in response to customer enquiries. You will need to find the information as quickly as possible, but may need to find a particular record from either the part number or the description. We shall therefore use the **find** command so that you can give any valid text to start the search.

The procedure must be able to ask for you to confirm that the record is the one you require. We shall delegate this task to a separate procedure, so we can use it in different situations if necessary.

```
proc confirm
  print : print "Confirm (y/n)";
  let yes=lower(getkey())="y"
  cls
  endproc
```

It leaves the variable **yes** containing 1 if you press the Y key – otherwise the value is zero. Note the use of the = sign for assignment and also in a logical condition.

```
proc inquire
  rem ***** inquire stock item *****
  print
  input "Stock item? "; name$
  use "sto"
  find name$
  let yes=0
  while found() and not yes
    display
    sprint
    confirm
    if not yes
     continue
     endif
    endwhile
  if not found()
    print
    print name$; " does not exist"
    endif
  endproc
```

This procedure merely locates the correct record. A more usable procedure for interrogating the stock file is **query**:

```
proc query
  inquire
  clear
  endproc
```

This uses another procedure, **clear**, which waits until you press a key, clears the screen and then prints a list of the commands you can use. We shall leave this procedure until we have written the procedures it must list. Remember to leave **edit** from time to time to save these procedures as you enter them.

Stock Report

We can also write a simple procedure to produce a general stock report.

```
proc report
  rem ***** stock report *****
  cls
  print tab 2; "ITEM"; tab 11; "CODE";
  print tab 20; "QUANTITY"; tab 31; "PRICE";
  print tab 40; "STOCK VALUE";
  print
  let total=0
  use "sto"
  all
    print description$( to 10);tab 11;sto.stockno$;
      tab 20;qty;
    print tab 31;"£";sellpr; tab 40;"£";sellpr*qty
    let total=total+sellpr*qty
    endall
  print
  print "Total stock value =£"; total
  clear
  endproc
```

Recording Sales

All we need to do to record a sale is to subtract the number of items sold from the relevant stock record. It is advisable to include some form of confirmation that we are dealing with the right stock item and that the stock is sufficient to meet the order.

```
proc quantity
   rem ***** print items in stock *****
   inquire
   print
   input "How many? "; num
   print
   cls
   print num;" * ";sto.stockno$;" (";sto.description$;")"
   endproc

proc sale
   rem ***** process sale *****
   quantity
   if num<=sto.qty
   print "Order value:- £"; num*sto.sellpr
   confirm
   if yes
      let sto.qty=sto.qty-num
      update
      sprint: rem *** show the modified record ***
      endif
   else
   print "Not enough stock"
   endif
clear
endproc
```

Recording Incoming Stock

The following procedure allows you to record the delivery of stock. Again it requests confirmation of the details you type in before accepting them and updating the relevant stock record.

```
proc delivery
   rem ***** in case stock on delivery *****
   quantity
   confirm
   print
   if yes
      print "Accepted"
      let sto.qty=sto.qty+num
      update
      sprint
      else
      print "Delivery not recorded"
      endif
   clear
   endproc
```

Ordering New Stock

So far our procedures have only referred to the stock file. When we want to order more stock we shall have to refer to the supplier and orders files for the name and address of the company, the price, and so on.

Assuming that we have identified the item in the stock file (with **inquire**) we select, from the orders file, those records that have the correct stock code. These records contain the codes for all the companies that can supply the item. Since the records also contain the price and delivery time for each supplier, we can decide whether we want the cheapest item or the shortest delivery time.

We use **locate** as a fast way of finding the required supplier record. This means that the supplier file must be ordered (with respect to the supplier code, code$) before we use **doorder**.

```
proc doorder
  rem *****order new stock *****
  inquire
  use "ord"
  select sto.stockno$=ord.stockno$
  print
  print "fast or cheap (f/c)";
  if lower(getkey())="f"
    fast
    else : cheap
    endif
  let ycode$=scode$
  reset
  use "sup"
  locate comp$
  doform
  print
  print "Expected delivery is ";del;" days"
  clear
  endproc
```

The procedure **cheap** finds the supplier with the lowest price, and **fast** works in the same way to find the supplier with the shortest delivery time.

```
proc cheap
  rem ***** find cheapest *****
  use "ord"
  let pri=price
  let comp$=code$
  let del=delivery
  all
    if price<pri
      let pri=price
      let comp$=code$
      let del=delivery
      endif
    endall
  endproc

proc fast
  rem ***** fastest delivery *****
  use "ord"
  let del=delivery
  let comp$=code$
  let pri=price
  all
    if delivery<del
      let del=delivery
      let comp$=code$
      let pri=price
      endif
    endall
  endproc
```

The procedure **doform** produces the actual order form. You should modify it to your own requirements. We shall use a simple version which shows the order details on the screen.

```
proc doform
  rem ***** produce order form *****
  cls
  print
  print sup.coname$
  print sup.street$
  print sup.county$
  print sup.postcode$
  print
  print "Please supply "; sto.buyqty;
```

```
print " * part number ";
print ycode$
print "("; sto.description$; ") ";
print "at £"; pri; " each."
print
print "Total value: £"; sto.buyqty*pri
endproc
```

The final command that we need is one to close all the files when we have finished using them.

```
proc bye
  confirm
  if yes
    cls
    print : print "bye"
    close "sto"
    close "sup"
    close "ord"
    cls
    endif
  endproc
```

We can now write a short procedure to run the application. It must open all three files with the correct logical file names, clear the display and show you the additional commands that you have. Note that, in normal use, the stock file is the only one whose records will need to be changed. The other two files are opened as read only files. It also orders the supplier file so that we can **locate** a company by its reference code.

```
proc start
  cls
  print at 5,5; "STOCK CONTROL DEMONSTRATION"
  print
  open "stock" logical "sto"
  look "supplier" logical "sup"
  look "orders" logical "ord"
  use "sup"
  order code$; a
  clear
  endproc
```

Finally we can write **clear**, which simply clears the screen and shows a list of the extra commands available:

```
proc clear
  rem  *****  clear  screen  and  get  command  *****'
  local x$
  print
  print "Press any key to continue ";
  let x$=getkey()
  cls
  print
  print "Query Report Delivery Doorder Sale Bye": print
  print "Type in your choice"
  endproc
```

CHAPTER 12
QL ARCHIVE REFERENCE

VARIABLES

Variable names may be up to thirteen characters in length, and must not start with a digit (0 to 9). They may contain any combination of upper or lower case alphabetic characters, or digits. Other characters are not allowed, except for $ and . which have special meanings.

If a variable name ends with a $ it is a string variable. Strings may be up to 255 characters in length. If the name does not end with a $ the variable is numeric. A variable name may refer to the contents of a record in a file and is then known as a field variable. Field variables are normally assumed to refer to the current file but may be made to refer to another open file by including a logical file name, separated by a . from the variable's name. Such a field variable is written as:

logical_file_name . field_name

For example **main.continent$**. If a variable name includes a dot then it must refer to a field in an open file. If there is no dot an attempt is made to match the name to an existing variable in the following sequence:

1 a field of the current file
2 a local variable (a parameter in the current procedure, if any)
3 a global variable

An error message is given if no match is found.

SYNTAX

The term *syntax* refers to the exact structure of a command or function. The syntax of a command specifies the parameters that the command needs, in what order they must appear, and the symbols (if any) used to separate them.

This section describes the notation used to express the syntax of Archive's programming language.

EXPRESSIONS

An *expression* is a combination of literal values, variables, functions and operators which results in a single value. A *numeric expression* results in a numeric value and a *string expression* results in a text value. Examples are:

```
3 * y * sin (x) + len (a$)      {numeric}
"abc" + a$ + rept (" - ", 5)    {string}
```

An expression may, as in the above examples, be composed of several sub-expressions. In such a case you may not mix sub-expressions of different types. They must all be string expressions or all numeric.

Syntax Conventions

The syntax definitions are similar to those used to define the syntax of SuperBASIC, i.e.:

Symbol	Meaning
italics	denotes a syntactic entity
[ ]	encloses an optional item
**	encloses items that may be repeated
|	or
{ }	comment

Syntactic Entities

s.lit	literal string
s.exp	string expression
n.exp	numeric expression
exp	expression, either string or numeric
ptm	print item
var	variable name, either string or numeric
lfn	logical file name
fnm	physical file name (up to 8 characters)
pnm	procedure name

A literal string is text enclosed in quotes, for example 'text' or "text".

A string expression is a literal string, or a combination of literal strings, string variables and string functions that results in a text value for example:

```
"fred"+a$+chr(72)
```

A numeric expression is either a number, or a combination of numbers, numeric variables and operators (+, −, *, /, etc) that results in a numeric value for example:

```
(3+x) / sin(y)
```

A print item is one of four possibilities: **at, tab, ink, paper.** A full description of a print item in our syntax notation is:

```
print_item:= | at n exp , n exp
             | tab n exp
             | ink n exp
             | paper n exp
```

Logical file names and procedure names have the same restrictions as variable names. Physical file names must also not exceed eight characters.

As an example of a *syntax definition,* consider the syntax of the order command. In our notation it appears as:

```
order spec:= var; a | d
order order spec *[ , order spec ]*
```

Order therefore needs to be followed by at least one *order specification* which itself consists of a *variable* separated by a colon from a letter which must be either **a** or **d.** In addition you can also include up to three further order specifications provided each pair is separated by commas. Clearly the syntax notation provides a much more compact description.

Note that the syntax notation does not tell you the meaning or purpose of the symbols so you will have to read the rest of the description for each command. The syntax only gives you a formal description of the number and kind of items that go to make up a valid command. In addition the syntax notation does not tell you the maximum number of repetitions allowed for the repeated items. **Order** will accept up to four pairs of a variable and a letter.

ARCHIVE DATA FILES

A Field

A *field* is the space reserved to hold either a string or a number.

In Archive, each field is identified by a field variable name. Whether a particular field can hold a string or a number is dependent on the name given to the field at the time it was created – string fields have a name ending with a **$**. An Archive string field may hold up to 255 characters. A numeric field has a name that does not end with a **$** sign. All numbers are stored in the same amount of space, regardless of their value. The possible range for a number is the same as the valid numeric range for the arithmetic operators.

A Record

A *record* is a collection of fields, whose contents are related in some way. The fields of a record might, for example, be used to hold the name, the address and the telephone number of a particular person. In Archive the records are of *variable length* so that each record only takes up as much room as is necessary to hold the information contained in its fields. There may be up to 255 fields in an Archive record.

A File

A *data file* is made up from a number of related records. To continue the above example, a data file could consist of a collection of name, address and telephone number records for many different people. The number of records in an Archive data file is limited to roughly 15 000. In practice, you are limited to the capacity of one Microdrive cartridge, which will hold about 1000 records of 100 characters. A file is the basic unit that you can save on, or load from, a Microdrive cartridge. Each file has a name to identify it. In Archive you give a physical name to the file when it is created, but you can change the logical name at any time.

Opening and Closing Files

When you want to read from or write to a data file you must first *open* it. Generally speaking, opening a data file transfers a copy of the file from the Microdrive cartridge into memory although, in the case of a long file, it is possible that only part of the file will be present in memory at any one time.

You can open a data file in *read only* mode with **look** which, as its name suggests, means that you can not change its contents. You also have the option of opening a data file in *update* mode with **open** so that you are allowed both to read and to change its contents.

Every time you open a data file, Archive reserves space for the field variables needed by a record within the file. The field variables always contain the values of the current record.

When you close a data file with **close** or **quit** any changes that you have made are copied into the file stored on the Microdrive cartridge. The copy held in memory is discarded. Closing a file is the only way of ensuring that the copy on the Microdrive cartridge contains your latest version. Since an open file uses part of the computer's memory, you should not leave files open if you are not using them.

When you leave Archive with the **quit** command, all open files are closed automatically.

Do not turn off the computer, or remove a cartridge from a Microdrive, while the cartridge contains open files.

Logical File Names

Each open data file has an associated *logical file name*, given to it when the file is opened. If you do not specify a logical file name when you open the file, it is automatically given the logical file name "main".

The logical file name is used to identify a particular file when you are using several files at once.

PROCEDURES

A procedure is a named section of program, starting with a procedure declaration of the form:

proc *pnm*[; *var* *[, *var*]*]

and ending with:

endproc

It may be referred to by name from any other program or procedure, including itself. It acts as though its code had been inserted at the point from which it is called.

In Archive, the **proc** and **endproc** commands cannot be entered directly at the keyboard, but are added automatically when you use the program editor to create a procedure.

THE PROGRAM EDITOR

The program editor is entered using the **edit** command.

If there are no procedures present in memory, you will be immediately offered the option of creating a new procedure. Otherwise you are given a list of all the procedures in memory on the left hand side of the display area. The first procedure is highlighted and is listed in full on the right hand side of the display. The first line of the procedure is highlighted to mark the current procedure and the current line.

Once in **edit** you have five options:

Select a procedure
Press **TABULATE** to move down the list of procedures, press **SHIFT** and **TABULATE** to move up the list. The listing on the screen always shows the current procedure.

Select a line
Use the up and down cursor keys to select a line within the current procedure. The current line is highlighted.

Press F3 for the menu of editing commands.
There are four commands, which are selected by pressing the key corresponding to the first letter.

Delete Press **ENTER** to delete the procedure highlighted on the left of the display. Press any other key to leave the command without deleting the procedure.

New Type in the name of the new procedure and press **ENTER**. If a procedure of that name already exists you will be offered the opportunity to edit it.

Cut Removes text from the current procedure and transfers it to the paste buffer. Before calling this command use the up or down cursor keys to make the first (or last) line of the region to be removed the current line. Then use the up and down cursor keys to mark the region of text to be removed. Press **ENTER** to remove the text into the paste buffer.

Paste Copy the contents of the paste buffer into the current procedure below the current line. **Paste** will clear the paste buffer.

Insert text
Press **F4** to insert one or more lines of text below the current line in the current procedure. Type the text and press **ENTER**. Pressing **ENTER** without any preceeding text will leave the insert option.

Edit text
Press **F5** to edit the current line of the current procedure. The line of text is copied into the input line and can be edited with the line editor. Press **ENTER** to replace the old line with the new line.

THE SCREEN EDITOR

The screen editor is entered with the **sedit** command. It allows you to design a new screen layout or modify an existing one. Once you have designed a layout you can save it on a Microdrive cartridge with the **ssave** command and load it with the **sload** command.

A screen layout is composed of two parts, the fixed background text and the variable values that are displayed in it. The **screen** command shows the background text and the **sprint** command adds the current values of the variables it contains.

Sedit has two options:

type text into the screen background
press **F3** to use a screen editing command.

There are four screen editing commands available after pressing **F3**:

C – clear the screen
V – mark a region to show a variable
I – set the ink colour
P – set the paper colour.

A screen layout is made active by:

```
sload
screen
```

When a particular screen is active it will show the current values of its variables after **sprint**, or when control returns to the keyboard after executing a program (or a command). A screen layout is made inactive by clearing the screen with **cls**. If there is no active screen, **sprint** has no effect. You may only have one screen layout in the computer's memory at any one time.

The **display** command creates and uses its own screen layout. It will therefore replace any other screen layout with its own design.

THE COMMANDS

The following commands are available.

ALL

Scans through the logically present records of the file in the fastest possible time.

Syntax: **all** [*lfn*] : ... : **endall**

This scan will not, in general, be in any particular sequence. The optional logical file name will force it to refer to a specified open file. If the logical file name is not given then it will scan the current file.

The **all** loop is primarily designed for examining the file records rather than for changing them. Do not use **update** within an **all** loop, unless you are sure that the length of the record will remain unchanged. You may, for example, change the value of a number, or convert a text field to upper case. If in doubt, use a **while** loop – using the value of **eof( )** to detect the end of the file. For example

```
first
while not eof( )
  ...
  update
  ...
  next
  endwhile
```

ALTER

Alters the current screen layout to display the current values of the variables.

Syntax: **alter**

You can change the contents of any one or more fields of the current file whose values are shown in the screen layout. Note that it is not necessary for all the field variables to be shown. You can not change a field that is not shown. If none of the field variables appear in the screen, Archive forces a **display** of the file.

First select the field to change by pressing **TABULATE** or **ENTER** until the cursor is at the correct field (variables that are not fields of the file are skipped). You can then type a new value or use the line editor to modify the existing value. Press **TABULATE** or **ENTER** to move to the next field. (Pressing **SHIFT** and **TABULATE** together moves back to the previous field.)

When you have made all the changes you want, press **F5** to replace the old record with the new one. The record is replaced automatically if you press **ENTER**. If the file is ordered the new version of the record is inserted in sequence.

APPEND

Adds a record to the specified file, or to the current file if the logical file name is not given.

Syntax: **append** [*lfn*]

The fields of the record take the current values of the field variables. If the file is ordered, the insertion is in sequence.

BACK

Moves backwards one record in the specified file, or in the current file if the logical file name is not given.

Syntax: **back** [*lfn*]

BACKUP

Makes a copy of the specified file. You should make copies of all your files, to protect against accidental damage or erasure.

Syntax: **backup** *oldfnm* **as** *newfnm*

CLOSE

Closes the specified file, or the current file if no logical file name is specified.

Syntax: **close** [*lfn*]

CLS

Clears the display area and switches off any display screen. See **screen**, **sload**, **sprint**.

Syntax: **cls**

CONTINUE

Continues the previous **search** or **find**, from the record following the current record in the current file.

Syntax: **continue**

CREATE

Creates a named open file whose records contain the fields given by the list of variables specified in the command. You have the option of specifying a logical file name – if you do not the file is created with the logical file name "main".

Syntax: **create** *fnm* [**logical:** *lfn*] : *var* *[: *var*]* : **endcreate**

DELETE

Deletes the current record from the specified file, or from the current file if no logical file name is given.

Syntax: **delete** [*lfn*]

Warning: Use this command with care since you can not recover the deleted record.

DIR

Displays a list of files on a Microdrive cartridge.

Syntax: **dir** [*drive*]

You may specify the Microdrive to be either **mdv1** or **mdv2**. If you do not include the Microdrive name Archive will automatically list the files on the cartridge in Microdrive 2.

Before showing the list of files, Archive displays the volume name of the cartridge (the name you gave when you formatted it).

DISPLAY

Shows the logical file name of the current file and a list of the field names and the values of the field variables for the current record. If the file is sorted, it also shows the sort fields and their sort priority.

Syntax: **display**

The command replaces any existing user-defined screen layout with this list, which becomes the active screen layout.

DUMP

Syntax: **dump** [; *var*] *[, *var*]*

Prints the specified fields of the selected records of the current file in tabular form **ser1** output. If you do not give a list of field variable names, *all* the fields are printed.

You can divert the output to a Microdrive file with **spoolon**.

EDIT

Calls the procedure editor to create a new procedure or to edit an existing procedure.

Syntax: **edit**

ENDALL

See **all**.

ENDCREATE

See **create**.

ERROR

Syntax: **error** *pnm*[; *exp* *[, *exp*]*]

Marks a procedure for the purposes of error-handling. Any error which occurs during the execution of this procedure, or any other procedure which it calls, causes a premature return from the marked procedure. The procedure can determine the nature of the error by using the **errnum( )** function to read the error number. This error number is cleared each time that **error** is executed.

EXPORT

Saves the named fields of the selected records of the current Archive file on a Microdrive cartridge in a form suitable for import to QL Abacus or QL Easel.

Syntax: **export** *fnm* [; *var*] *[, *var*]* [**quill**]

If you do not specify a list of field variable names, *all* the fields are exported. If you include the optional parameter **quill**, (separated by at least one space from the last variable name) the file is exported in a form suitable for import by QL Quill.

The export file is named *fnm* and, unless you specify your own file name extension, Archive uses the extension **_EXP**.

See the *Information section* for a full discussion of import and export.

FIND

Rewinds the file to the beginning and searches for the first record containing a match to the specified string in any string field. The match is independent of upper or lower case text.

Syntax: **find** *s.exp*

You can continue the search with the **continue** command, and determine whether the search was successful by examining the value returned by the **found( )** function.

FIRST

Finds the first record of the specified file, or the current file if no logical file name is specified.

Syntax: **first** [*lfn*]

FORMAT

Formats the cartridge in Microdrive 2 (the right hand drive). It gives the cartridge the name you specified. This name is reported when you subsequently use **dir** to show a directory of the files on that cartridge.

Syntax: **format** "*you specified*"

IF Allows a specified condition to control subsequent processing.

Syntax: **if** *n.exp* : ... [: **else** : ...] : **endif**

Without **else**.
If the expression is non-zero, the following statements are executed. If the expression is zero execution transfers to the statement following **endif**.

With **else**.
If the numeric expression is non-zero, the statements between **if** and **else** are executed. Otherwise the statements between **else** and **endif** are executed. In either case execution continues with the statements following **endif**.

IMPORT Reads a file, *name1*, exported from QL Abacus or QL Easel and produces an Archive data file *name2*. As with **open** and **look** you have the option of specifying a logical file name for the data file.

Syntax: **import** *name1* **as** *name2* [**logical** *lfn*]

where: *name1*:= *fnm*
name2:= *fnm*

See the *Information section* for a full description of import and export.

INK Sets the foreground colour for all following text to the colour specified by the value of the expression.

Syntax: **ink** *n.exp*

The colours are: 0 and 1 black
2 and 3 red
4 and 5 green
6 and 7 white

If the expression evaluates to more than 7, the value taken is the remainder after division by 8, for example **ink** 9 is equivalent to **ink** 1, both setting the print colour to black. If ink is used within a **print** command it will only change the print colour for the duration of that command.

INPUT Requests input from the keyboard to the variables listed in the command. Each variable in an input list may be preceded by a initial string which will be displayed as a prompt for the input. All input items must be separated from each other by semicolons. If the list has a final semicolon, the cursor will not move to a new line after the input.

Syntax: **input** [*var* | *s.lit* | *ptm* ***** [; *var* | *s.lit* | *ptm*]*****][;]

The list of input items may include the cursor-positioning items

at *line,column*
tab *column*

where: *line*:= *n.exp*,
column:= *n.exp*

The first of these positions the cursor at the specified line and column position, and **tab** moves the cursor to the specified column within the current line. If the cursor is already to the right of the specified column, tab will have no effect.

These two items may not be used outside an **input** or a **print** command.

You may also use ink and paper as input items. If used within an input command they will only affect the ink and paper colours to the end of the input, when the colours will return to their original settings.

INSERT Adds a new record to a file.

Syntax: **insert**

Uses the current screen layout to display the current values of the variables. You can type a new value for any one or more fields of the current file whose values are shown in the screen layout. Note that it is not necessary for all the field variables to be shown. You cannot type a value for a field that is not shown. If none of the field variables appear in the screen, Archive forces a **display** of the file.

First select a field by pressing **TABULATE** or **ENTER** until the cursor is at the correct field (values that are not fields of the file are skipped). You can then type a new value. Press **TABULATE**or **ENTER** to move to the next field. (Pressing **SHIFT** and **TABULATE** together moves back to the previous field.)

When you have typed all the values you want you should press **F5** to add the new record to the file. The record will also be added to the file if you press **ENTER** when the cursor is in the last field. Any field that you have not given a value will be zero (if it is a numeric field) or an empty string (if it is a text field). If the file is ordered, the new record is inserted in sequence, otherwise the insertion takes place at an unspecified position.

KILL

Erases the specified file from the Microdrive cartridge.

Syntax: **kill** *fnm*

Warning: Use this command with care since you cannot recover the erased file.

LAST

Finds the last record of the specified file, or the current file if you do not specify a logical file name.

Syntax: **last** [*lfn*]

LET

Used to assign a value to a variable (as in SuperBASIC).

Syntax: **let** *var* = *exp*

LLIST

Lists all the procedures currently in memory on a printer.

Syntax: **llist**

LOAD

Loads the specified procedure file from a Microdrive cartridge into memory.

Syntax: **load** [**object**] *fnm*

LOCAL

If you include the optional **object** Archive will expect the file to be in binary rather than ASCII form, see **save**.

Within a procedure, forces the following list of variables to be local variables. These variables exist only within the procedure in which they are declared and are undefined in any other procedure. Their values are destroyed on exit from the procedure.

Syntax: **local** *var* *[, *var*]*

LOCATE

Finds, in an ordered file, the first record whose field contents match the expression(s).

Syntax: **locate** *exp* * [,*exp*]*

The record is located much more quickly than if you used **find**, but **the file must first have been sorted**. Each expression must explicitly refer to the contents of a particular sort field. In the case of a string field the match is case-dependent.

If you have ordered the file with respect to more than one field, you can specify several expressions (one for each sort field). The expressions are separated by commas and must refer to the fields used to order the file. They must be in the same sequence as in the preceding **order** command. For example:

```
order animal$ ; a , weight ; a
locate "Elephant" , 2000
```

will find the first record in which the field animal$ contains the text "Elephant" and a weight that equals (or exceeds) 2000.

If there is not an exact match **locate** will still find a record. This record will be the first one whose field contents "exceed" – in the sense of the ordering (i.e. "d" comes after "e" if the file is sorted in descending order) – the specified values.

LOOK

Opens the named file for read access only. If the logical file name is not specified, it is given the default value "main".

Syntax: **look** *fnm* [**logical** *lfn*]

LPRINT

Displays the values of the following list of items on a printer attached to SER1, in the same way as for **llist.**

Syntax: **lprint** [*exp* | *ptm* *[; *exp* | *ptm*]*] [;]

MERGE

Adds the procedures of the specified program file to the procedures already in the computer's memory. If the file contains a procedure with the same name as one already in memory, the new procedure replaces the old one.

Syntax: **merge** [**object**] *fnm*

If you include the optional **object** Archive will export the file to be a binary rather than ASCII format. See **Save**.

MODE

Changes the form of the display.

Syntax: **mode** *var,var*

The first variable may have a value of 0 or 1. A value of 0 joins the control, display and work areas into a single region. A value of 1 separates them back into three distinct areas.

The second variable may have a value of 4, 6 or 8 and switches the display between showing 40, 64 or 80 characters per line.

The initial setting, when you load Archive for use with a monitor, is equivalent to:

```
mode 1,8
```

NEW

Deletes all the data from the computer's memory, ready for a fresh start. Any open files are closed. (The command does not delete files stored on a Microdrive cartridge.)

Syntax: **new**

NEXT

Moves to the next record in the specified file, or in the current file if you do not specify a logical file name.

Syntax: **next** [*lfn*]

OPEN

Opens the specified file for both reading and writing. The file is given a logical file name "main" if you do not specify one.

Syntax: **open** *fnm* [**logical** *lfn*]

ORDER

Orders the records of the file according to the contents of the specified fields.

Syntax: **order** *order_spec* *[, *order_spec*]*

where: *order_spec*:= *var*; **a** | **d**

The first field specified in the list is the primary sort field. Records which have equal contents of their primary sort field are further sorted according to the contents of the next field in the list (if it is specified) and so on. For each specified field an ordering direction must be given. This must be either **a** or **d** to specify ascending or descending order respectively.

Order only takes account of the first 8 characters of a text field and you may not specify more than four fields to **order** the file.

PAPER

Sets the background colour for all following text to the colour specified by the value of the expression.

Syntax: **paper** *n.exp*

The colours are:

0 and 1 black
2 and 3 red
4 and 5 green
6 and 7 white

If the expression evaluates to more than 7, the value taken is the remainder after division by 8, i.e. **paper** 11 is equivalent to **paper** 3, both setting the colour to red.

If **paper** is used within a **print** command, it will only change the background colour for the duration of that command.

POSITION

Makes the record whose record number is given by the expression the current record.

Syntax: **position** *n.exp*

PRINT

Displays the values of the following list of items – which must be separated by semicolons – on the screen. If the list has a final semicolon, the cursor will not move to a new line after the display. See also **lprint**.

Syntax: **print** [*exp* | *ptm*]*[**;** *exp* | *ptm*]*] [**;**]

QUIT

Closes all files and returns to SuperBASIC.

Syntax: **quit**

REM

When used within a procedure, it marks the rest of the line as containing a comment. Any following text on that line is ignored when the procedure is executed.

Syntax: **rem**

RESET

This command restores all the records in the current file which were removed by an earlier use of **select**. It destroys any ordering of the file.

Syntax: **reset**

RETURN

Used within a procedure to cause an immediate termination of the procedure by returning to the calling procedure.

Syntax: **return**

RUN

Loads the specified procedure file into memory and starts execution of the procedure called **start**.

Syntax: **run** [**object**] *fnm*

SAVE

If you include the optional **object** Archive will expect the file to be in binary rather than ASCII form, see **save**.

Saves all procedures currently in memory as a single named file on a Microdrive cartridge.

Syntax: **save** [**object**] *fnm*

If you include the optional **object**, Archive will save the file in binary, rather than ASCII, format. This means that Archive does not have to convert the program into ASCII characters before saving it and is therefore much faster. You can **load, run** or **merge** such a program by adding the optional **object** to the appropriate command. These operations will also work more rapidly since no conversion is necessary. Such files have an extension of _pro, rather than the normal prg.

You may also save such an **object** program in a form that is protected against examination or modification. Include, instead of **object**, the optional **protect**. A program saved in this way can only be loaded, run or merged – using the optional **object** with the appropriate command.

A protected program cannot be listed, edited or saved. If you merge a protected program with any other program then the combination will be similarly protected. The only way to clear the protected status is with the **new** command.

Saving a protected version does not affect the copy of the program in the computer's memory. You can still list, edit or save the program in the normal way.

SCREEN

Displays the formatted screen layout previously sloaded. It does nothing if there is no screen layout present. It does not display any of the variables in the screen.

Syntax: **screen**

SEARCH Searches the current file from the beginning until a record is found in which the specified expression is true. This record becomes the current record.

Syntax: **search** *n.exp*

SEDIT Calls the screen editor, to enable you to define a new screen layout. See Chapter 7.

SELECT Scans the whole file selecting only those records for which the specified expression is true. The file then behaves as if only the selected records are present.

Syntax: **select** *n.exp*

You can restore all the discarded records with the **reset** command.

SINPUT Waits for input to the variables in the following list, using the order specified in the list. All the variables in the list must be currently displayed in an active screen layout.

Syntax: **sinput** *var* *[, *var*]*

SLOAD Loads a previously defined and saved display screen layout. It also displays this screen layout and activates the display of any variables within the screen.

Syntax: **sload** *fnm*

The displayed values are then updated automatically whenever control returns from a procedure to the keyboard interpreter.

SPOOLOFF Direct all following **lprint** and **llist** output to the printer. This cancels the effect of **spoolon**.

Syntax: **spooloff**

SPOOLON Directs all following **lprint, llist** and **dump** output to the specified file – or to the screen – instead of to the printer.

Syntax: **spoolon** <*fnm*> [**export** | *dump*]
or:
spoolon screen

If you are directing output to a file, it is directed via the currently installed printer driver so that it contains all the special codes that your printer needs.

If you include the optional **export**, Archive ensures that the file contains only printable ASCII codes, carriage returns and line feeds. The resulting file is suitable for importing into Quill.

The optional **dump** allows the text to be transmitted to the file without being processed by the printer driver. In this case all ASCII codes (including control codes) are passed straight into the file.

Unless you specify a file name extension, Archive assumes an extension of **_lis** (**_exp** or **_dmp** if you include the optional **export** or **dump**).

The alternative form of the command – **spoolon screen** – directs the output to the monitor screen instead of the printer.

SPRINT Used within a procedure to force a display of the fields of the current record.

Syntax: **sprint**

There must be an active screen layout (the screen layout is made active by a previous use of **screen**, **sload** or **display**). If there is no active screen layout, the command will have no effect.

SSAVE Saves, as a named file on a Microdrive cartridge, the current display area as a defined screen layout.

Syntax: **ssave** *fnm*

It saves the text of the screen and a list of the variables in the display, together with their positions.

STOP Terminates the execution of all procedures and returns control to the keyboard.

Syntax: **stop**

TRACE

Switches the trace mode on and off.

Syntax: **trace**

Type:

```
trace
```

to turn on the trace. In trace mode each line of the program is displayed in the work area of the screen, as it is executed. Press the space bar and keep it held down to pause. The trace will continue when you release the space bar. To turn the trace off again, type:

```
trace
```

UPDATE

Replaces the current record in the specified file (or the current file if no logical file name is given) with a record containing the current values of the field variables.

Syntax: **update** [*lfn*]

USE

Makes the specified file the current file.

Syntax: **use** *lfn*

WHILE

Repeatedly executes the statements between **while** and **endwhile** for as long as the value of the expression is non-zero (true).

Syntax: **while** *n.exp* : ... : **endwhile**

FUNCTIONS

Think of a function as a kind of recipe which converts one or more initial values, known as the function's *arguments*, into a different value, which is said to be the value that is *returned* by the function.

The functions provided by Archive may take three, two, one or no arguments. The arguments for a function are placed in brackets after its name. You must not leave a space between the name and the opening bracket, but spaces are allowed between items within the brackets. If a function takes more than one argument, the arguments are separated by commas. All functions must be followed by the brackets, even if they take no arguments. The presence of the brackets is a useful reminder that you are referring to a function. They allow you to distinguish between a variable and a function, even if they have the same name.

The following functions are provided.

ABS(*n.exp***)** Returns the absolute value of the argument, i.e. ignores any minus sign.

ATN(*n.exp***)** Returns the angle, in radians, whose tangent is *n.exp*.

CHR(*n.exp***)** This function returns the ASCII character whose code is n.exp. A character with an ASCII code less than 32 is only sent to the printer if preceded by an ASCII null. For example:

```
lprint chr(0)+chr(13)
```

passes the ASCII character for a carriage return to a printer. This is useful if your printer needs control code sequences to produce special effects – refer to your printer manual for any special codes that it needs.

You can, for example, send an "A" to the screen with:

```
print chr(65).
```

CODE(*s.exp***)** This returns the ASCII value of the first character found in the specified text.

COS(*n.exp***)** Returns the cosine of the given (radian) angle.

COUNT([*lfn* **])** Returns the count of the number of records in the current file.

DATE(*n.exp***)** Returns today's date as a text string in one of three forms:

n.exp	date string
0	"YYYY/MM/DD"
1	"DD/MM/YYYY"
2	"MM/DD/YYYY"

You must first have set the system clock, as described in the *SuperBASIC Keyword Guide*.

DAYS(*s.exp*) Returns a number of days, from the first of January 1583, to a date given as a text expression of the form "YYYY/MM/DD". The conversion assumes the Gregorian (modern) calendar is being used. The formula is therefore only valid for dates after 1582.

DEC(*value,dp,width*)
value:=(n.exp)
dp:= (n.exp)
width:= (n.exp)

Converts the given numeric *value* to the equivalent text string, in decimal format with *dp* decimal places. The text is justified right in a field of *width* characters. For example:

dec(1.23e1,3,10) returns the text " 12.300" (with 4 leading spaces).

DEG(*n.exp*) Takes an angle, measured in radians, and converts it to the same angle in degrees.

EOF([*lfn*]) Returns a value indicating whether you have attempted to read past the end of the current file, or the specified file if a file identifier is given. The value returned is 1 if you have attempted to read past the end of the file, otherwise it is zero.

ERRNUM() Returns the number of the last error which occurred (an error number of zero indicates no errors). The error number is the same as that displayed together with the error message when Archive reports a detected error.

EXP(*n.exp*) Returns the value of e (approximately 2.718) raised to the power of (*n.exp*). The returned value will be in error if *n.exp* is greater than +88 since the result will then exceed the numeric range of Archive.

FIELDN(n.exp|, *lfn*|)
Returns the name of the specified field in the current record of the specified file (or the current file if no logical file name is given). Note that **fieldn(0)** returns the name of the first field.

FIELDT(*n.exp* [, *lfn*])
Returns the type of the specified field in the current record of the specified file (or the current file if no logical file name is given). Note that **fieldt(0)** returns the type of the first field.

It returns the value 0 if the field is numeric, otherwise it returns 1.

FIELDV(*n.exp*[, *lfn*])
Returns the value of the specified field in the current record of the specified file (or the current file if no logical file name is given). Note that **fieldv(0)** returns the value of the first field.

FOUND() Returns one if a record is found by use of **search** or **find**, otherwise returns zero.

GEN(*value,width*)
value:=n.exp
width:=n.exp

Converts the given numeric *value* to the equivalent text string, in general format. The text is justified right in a field of *width* characters. For example:

gen(1.23e1,10)

returns the text " 12.3" (with 6 leading spaces).

GETKEY() Waits for a key to be pressed and returns a single text character which corresponds to the key that was pressed.

INKEY() Returns the single text character corresponding to any key that was being pressed at the time the function is called. It does not wait for a keypress, but will return a null string ("") if no key is pressed.

INSTR(*main,sub)*

main:= *s.exp*
sub:= *s.exp*

This finds the first occurrence of *sub* within *main* and returns the position of the first character of sub in main. It will return a value of zero if no match is found. The match is case-dependent.

```
instr("January","Jan")    {returns 1}
instr("January","an")     {returns 2}
instr("January","AN")     {returns 0}
```

INT(*n.exp*) Returns the integer value of the number, by truncating at the decimal point. The truncation always operates towards zero. Thus;

```
int(3.7)      {returns 3}
int(-4.8)     {returns -4}
```

LEN(*s.exp*) Returns the number of characters in the specified text.

LN(*n.exp*) Returns the natural, or base e, logarithm of n.exp. An error results if n.exp is negative or zero, since logarithms are not defined in this range.

LOWER(*s.exp*) Converts the specified text to lower case.

MEMORY( ) Returns the number of unused bytes of memory remaining.

MONTH(*n.exp*) Returns, as text, the name of a month.

For example **month(3)** returns the text "March".

If an argument larger than 12 is used, it is replaced by the remainder after division by 12 so that, for example, **month(13)** and **month(1)** will both give the result "January".

NUM(*value, width*)

value:= *n.exp*
width:= *n.exp*

Converts the given numeric *value* to the equivalent text string, in integer format. The text is justified right in a field of *width* characters. For example:

num(1.23e1,10) returns the text " 12" (with 8 leading spaces).

NUMFLD([*lfn*]) Returns the number of fields in the records of the specified file (or the current file if you do not give a logical file name).

PI() Returns the value of the mathematical constant π.

RAD(*n.exp*) Takes an angle, measured in degrees, and converts it to the same angle in radians.

RECNUM([*lfn*]) Returns the number (counting from zero at the first record) of the current record of the specified file (or the current file if you do not give a logical file name).

REPT(*s.exp,n.exp*)

This function returns a string consisting of a number of copies of the first character of the given text. The resulting text may be up to 255 characters in length. For example,

```
print rept("*",5)       {will print five asterisks}
print rept("abc",3)     {prints "aaa"}
```

SGN(*n.exp*) Returns +1, −1 or 0, depending on whether the argument is positive, negative or zero.

SIN(*n.exp*) Returns the value of the sine of the specified (radian) angle.

SQR(*n.exp*) Returns the square root of the argument, which must not be negative.

STR(*n,type,dp*) *n*:= *n.exp*
type:= *n.exp*
dp:= *n.exp*

Converts a number, n, to the equivalent text string.

The second parameter, type, indicates the form of the converted string as follows;

0 decimal (floating point)
1 exponential, or scientific, notation
2 integer
3 general format

The third parameter, dp, indicates the number of figures after the decimal point in the converted string. It should always be specified, although its value is ignored for integer and general formats.

For example:

```
let a$=str(12.3456,0,2)     {gives a$ the value "12.35"}
let a$ str(12.3456,1,4)     {gives a$ the value "1.2346e1"}
```

TAN(*n.exp*) Returns the tangent of the specified (radian) angle.

TIME( ) Returns, as text, the time of day in the format "HH:MM:SS". You must first have set the system clock, as described in the SuperBASIC Keyword Guide.

UPPER(*s.exp*) Converts the specified string to upper case.

VAL(*s.exp*) Converts the text to its equivalent numeric value. It will only convert text composed of valid numeric characters and the conversion will stop at the first character that can not be interpreted as a digit. For example, **val("1.1ABC")** will return the numeric value l.l, and **val("ABC")** will return 0.0

VALUE(*s.exp*) Returns the value of the variable whose name is given by *s.exp* – for example:

```
let a$=''len''
let length=15
print value(a$+''gth'')
```

will print the value 15.

Note that value(fieldn(y)) is exactly equivalent to fieldv(y).

ERRORS

When ARCHIVE detects an error in a command typed at the keyboard or in a procedure, it displays an error number and a short error message. Examples of errors that would be detected are:

attempting to divide by zero
if not matched with an **endif**
supplying a procedure with the wrong number of parameters.

If the error comes from keyboard input, the text of the statement remains visible in the work area. You can press **F5** to recall the text so that you can use the line editor to correct the error. You can then press **ENTER** to execute the corrected statement.

If the error comes from a program statement, ARCHIVE shows the name of the procedure and the line in which the error occurred. You can then use the program editor to correct the error.

When you use the **error** command in your programs, ARCHIVE will not report any error that it detects in a procedure marked with **error**. You are free to deal with any such error in any way that you want (including ignoring it). You can find which error has occurred by examining the value returned by **errnum( )**. This number is the same as the one ARCHIVE gives when it prints an error message.

The following list shows ARCHIVE's error numbers, together with the corresponding messages. Where possible, the list includes a short example of a statement that would give the error. The error messages are not designed to pinpoint the precise error, but are intended to give you an idea of what type of error to look for.

Those error messages for which there is no short example are marked with an asterisk. They are dealt with in the notes which follow the list.

No.	Message	Example
0	no error	
1	command not recognized	apend
2	end of statement expected	let x=3 let y=4
3	variable name expected	let 31=x
4	unrecognized print item	print create
5	wrong data type	* (1)
6	numeric expression expected	let x="fred"
7	string expression expected	let x$=4
8	variable not found	let x=qq (qq undefined)
9	variable undefined	print qq
10	missing separator	print at 5
11	name too long	let thisverylongname=4
12	duplicate name	create:n$:n$:endcreate
13	string literal expected	* (2)
14	missing endproc	* (3)
15	bad proc statement	* (3)
16	premature end of statement	create"test":endcreate
17	program structure fault	* (4)
18	too many numbers	* (5)
50	missing closing quote	let x$="fred
51	missing exponent after "E"	let x=1.2E
52	number too big	let x=1.2E100
53	unknown symbol	let x=%
70	evaluator syntax error	let x=3+
71	mismatched parenthesis	let x=(3+5)/7)
73	type mismatch	let x$="fred"+3
74	wrong number of arguments	let x$=str(1,2)
75	string too long	let x$=rept("*",256)
76	divide by zero	let a=0: let x=5/a
77	bad function arguments	let x$=sqr(-4)
78	string subscript error	let x$="fred" (to 97)
80	out of memory	* (6)
90	no room to open a file	* (7)
91	incomplete file transfer	* (8)
93	out of range	print at 100,100;37
94	file not open	append (without first opening a file)
100	cannot open file	look"xxx" (non-existent)
101	write to read only file	look "names":insert
103	wrong file type	sload"names" (data file)
104	bad file name	save"3test"
105	error reading file	* (9)

Notes

1) The most likely cause of error 5 – "wrong data type" – is that you have inputted text when a number is expected, e.g. in response to an **input** statement such as:

```
input x
```

2) Error 13 – "string literal expected" – can occur, for example, during the import of a file that you have constructed yourself (without using any of the **export** commands in the QL programs). It means that Archive has found a number, or a numeric or text expression, where it was expecting to find a literal text value. In most situations where Archive finds numeric data when expecting text, or vice versa, it will give error 7 or error 8.

3) Errors 14 – "missing endproc" – and 15 – "bad proc statement" – should never occur in normal use. They indicate that Archive has detected a missing **endproc** or an error in the structure of a **proc** statement in a procedure. They are only likely to occur if you construct a program file with an editor other than the one included in Archive.

4) Error 17 – "program structure fault" – usually indicates that an **all**, **if** or **while** is not paired with a corresponding **endall**, **endif** or **endwhile** in a procedure. You

can also generate this error by including an **endproc** inside another program structure, or by using **return** directly from the keyboard.

5) Error 18 – "too many numbers" – indicates that you are trying to input more numbers than will fit into the memory reserved for input. The error may occur either in a line of input from the keyboard, or while loading a program that includes a procedure with many numbers in one of its lines. The exact limit depends on circumstances – a typical limit would be 15 to 20 numbers, so you are unlikely to get this error.

6) Error 80 – "out of memory" – should only be given if you use a very large program The size of an ordinary data file is not limited by the amount of memory in the computer since only part of a large file is in memory at any one time. If Archive gives you this error you will have to reduce the size of your program before continuing. You can, if necessary, break your program into several sections, in different files, and use **merge** to load each section as it is needed. This technique will, however, normally need a considerable amount of programming skill.

7) Error 90 – "no room to open a file" – occurs when the area of memory Archive reserves to store internal information about the files currently in memory becomes full. This may happen even if there is still memory available (i.e. if the value returned by memory() is still not close to zero.)

8) Error 91 – "incomplete file transfer" – means that the loading or saving of a file has failed for some reason. This may mean that the data has been corrupted, or that the cartridge or the Microdrive has been damaged.

9) Error 105 – "error reading file" – means that some of the data in a file is in the wrong format, the wrong order, or has been corrupted. This is only likely to occur if you construct your own import file – or your own program file without using the Archive program editor (advanced uses).